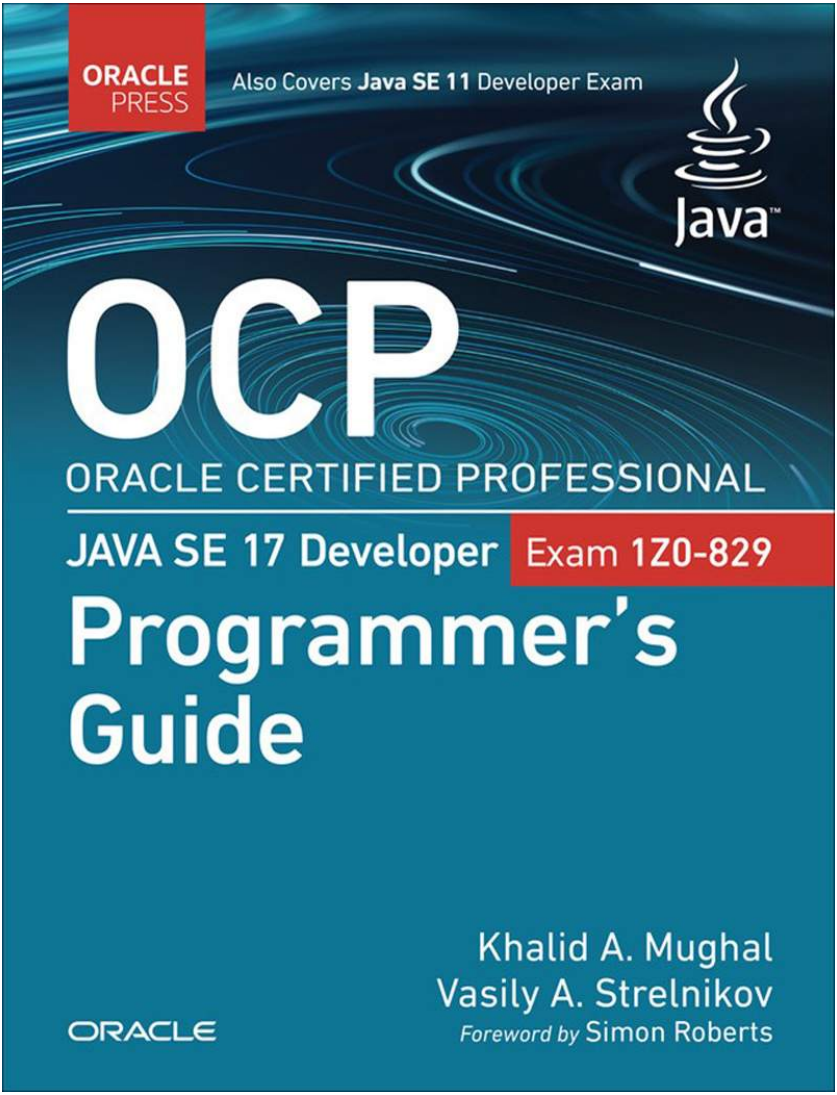


ORACLE PRESS

Also Covers **Java SE 11** Developer Exam

OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 17 Developer Exam 1Z0-829

Programmer's Guide

Khalid A. Mughal
Vasily A. Strelnikov
Foreword by Simon Roberts

ORACLE

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

OCP
Oracle Certified Professional
Java SE 17 Developer (Exam 1Z0-829) Programmer’s Guide

Pearson

OCP
Oracle Certified Professional Java SE 17 Developer (Exam 1Z0-829) Programmer's Guide

Khalid A. Mughal
Vasily A. Strelnikov

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: your/Shutterstock

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The views expressed in this book are those of the author and do not necessarily reflect the views of Oracle.

Oracle America Inc. does not make any representations or warranties as to the accuracy, adequacy or completeness of any information contained in this work, and is not responsible for any errors or omissions.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com

Library of Congress Control Number: 2022951639

Copyright © 2023 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.xhtml.

To the loving memory of my mother, Zubaida Begum, and my father,
Mohammed Azim.
And to the future generation: Tobias Albert, Ronja Johanne and Serine
Begum—with all my love.
—K.A.M.

Contents Overview

Figures

Tables

Examples

Foreword

Preface

1 Basics of Java Programming

2 Basic Elements, Primitive Data Types, and Operators

3 Declarations

4 Control Flow

5 Object-Oriented Programming

6 Access Control

7 Exception Handling

8 Selected API Classes

9 Nested Type Declarations

10 Object Lifetime

11 Generics

12 Collections, Part I: `ArrayList<E>`

13 Functional-Style Programming

14 Object Comparison

15 Collections: Part II

16 Streams

17 Date and Time

18 Localization

19 Java Module System

20 Java I/O: Part I

21 Java I/O: Part II

22 Concurrency: Part I

23 Concurrency: Part II

24 Database Connectivity

25 Annotations

26 Secure Coding

A Taking the Java SE 17 and Java SE 11 Developer Exams

B Exam Topics: Java SE 17 Developer

C Exam Topics: Java SE 11 Developer

D Annotated Answers to Review Questions

E Mock Exam: Java SE 17 Developer

F Annotated Answers to Mock Exam

G Java Logging API Overview

Index

Contents

Figures

Tables

Examples

Foreword

Preface

- Writing This Book
- About This Book
- Using This Book
- Book Website
- Request for Feedback
- About the Authors
- Acknowledgments

1 Basics of Java Programming

- 1.1 The Java Ecosystem
- 1.2 Classes
- 1.3 Objects
- 1.4 Instance Members
- 1.5 Static Members
- 1.6 Inheritance
- 1.7 Aggregation
- *Review Questions*
- 1.8 Sample Java Program
- 1.9 Program Output
- *Review Questions*

2 Basic Elements, Primitive Data Types, and Operators

2.1 Basic Language Elements
2.2 Primitive Data Types
2.3 Conversions
2.4 Type Conversion Contexts
2.5 Precedence and Associativity Rules for Operators
2.6 Evaluation Order of Operands
2.7 The Simple Assignment Operator `=`
2.8 Arithmetic Operators: `*`, `/`, `%`, `+`, `-`
2.9 The Binary String Concatenation Operator `+`
2.10 Variable Increment and Decrement Operators: `++`, `--`
Review Questions
2.11 Boolean Expressions
2.12 Relational Operators: `<`, `<=`, `>`, `>=`
2.13 Equality
2.14 Boolean Logical Operators: `!`, `^`, `&`, `|`
2.15 Conditional Operators: `&&`, `||`
2.16 Integer Bitwise Operators: `~`, `&`, `|`, `^`
2.17 Shift Operators: `<<`, `>>`, `>>>`
2.18 The Conditional Operator `?:`
2.19 Other Operators: `new`, `[]`, `instanceof`, `->`
Review Questions

3 Declarations

3.1 Class Declarations
3.2 Method Declarations
3.3 Statements
3.4 Variable Declarations
3.5 Instance Methods and the Object Reference `this`
3.6 Method Overloading
3.7 Constructors
3.8 Static Member Declarations
Review Questions
3.9 Arrays
3.10 Parameter Passing

- 3.11 Variable Arity Methods
- 3.12 The `main()` Method
- 3.13 Local Variable Type Inference
- *Review Questions*

4 Control Flow

- 4.1 Selection Statements
- 4.2 The `switch` Statement
- 4.3 The `switch` Expression
- *Review Questions*
- 4.4 Iteration Statements
- 4.5 The `while` Statement
- 4.6 The `do-while` Statement
- 4.7 The `for(;;)` Statement
- 4.8 The `for(:)` Statement
- 4.9 Transfer Statements
- 4.10 Labeled Statements
- 4.11 The `break` Statement
- 4.12 The `continue` Statement
- 4.13 The `return` Statement
- *Review Questions*

5 Object-Oriented Programming

- 5.1 Implementing Inheritance
- 5.2 The Object Reference `super`
- 5.3 Chaining Constructors Using `this()` and `super()`
- *Review Questions*
- 5.4 Abstract Classes and Methods
- 5.5 Final Declarations
- *Review Questions*
- 5.6 Interfaces
- *Review Questions*
- 5.7 Arrays and Subtyping
- 5.8 Reference Values and Conversions
- 5.9 Reference Value Assignment Conversions

5.10 Method Invocation Conversions Involving References

5.11 Reference Casting and the `instanceof` Operator

5.12 Polymorphism

Review Questions

5.13 Enum Types

5.14 Record Classes

5.15 Sealed Classes and Interfaces

Review Questions

6 Access Control

6.1 Design Principle: Encapsulation

6.2 Java Source File Structure

6.3 Packages

6.4 Searching for Classes on the Class Path

Review Questions

6.5 Access Modifiers

6.6 Scope Rules

6.7 Implementing Immutability

Review Questions

7 Exception Handling

7.1 Stack-Based Execution and Exception Propagation

7.2 Exception Types

7.3 Exception Handling: `try`, `catch`, and `finally`

7.4 The `throw` Statement

7.5 The `throws` Clause

Review Questions

7.6 The Multi-`catch` Clause

7.7 The `try`-with-resources Statement

7.8 Advantages of Exception Handling

Review Questions

8 Selected API Classes

8.1 Overview of the `java.lang` Package

- 8.2 The `Object` Class
- 8.3 The Wrapper Classes
- *Review Questions*
- 8.4 The `String` Class
- 8.5 The `StringBuilder` Class
- *Review Questions*
- 8.6 The `Math` Class
- 8.7 The `Random` Class
- 8.8 Using Big Numbers
- *Review Questions*

9 Nested Type Declarations

- 9.1 Overview of Nested Type Declarations
- 9.2 Static Member Types
- 9.3 Non-Static Member Classes
- *Review Questions*
- 9.4 Local Classes
- 9.5 Static Local Types
- 9.6 Anonymous Classes
- *Review Questions*

10 Object Lifetime

- 10.1 Garbage Collection
- 10.2 Reachable Objects
- 10.3 Facilitating Garbage Collection
- 10.4 Invoking Garbage Collection Programmatically
- *Review Questions*
- 10.5 Initializers
- 10.6 Field Initializer Expressions
- 10.7 Static Initializer Blocks
- 10.8 Instance Initializer Blocks
- 10.9 Constructing Initial Object State
- *Review Questions*

11 Generics

11.1 Introducing Generics
11.2 Generic Types and Parameterized Types
11.3 Collections and Generics
11.4 Wildcards
11.5 Using References of Wildcard Parameterized Types
11.6 Bounded Type Parameters
11.7 Generic Methods and Constructors
11.8 Implementing a Simplified Generic Stack
Review Questions
11.9 Wildcard Capture
11.10 Flexibility with Wildcard Parameterized Types
11.11 Type Erasure
11.12 Implications for Overloading and Overriding
11.13 Limitations and Restrictions on Generic Types
Review Questions

12 Collections, Part I: `ArrayList<E>`

12.1 Lists
12.2 Declaring References and Constructing `ArrayList`s
12.3 Modifying an `ArrayList<E>`
12.4 Querying an `ArrayList<E>`
12.5 Iterating Over an `ArrayList<E>`
12.6 Converting an `ArrayList<E>` to an Array
12.7 Creating List Views
12.8 Arrays versus `ArrayList`s
Review Questions

13 Functional-Style Programming

13.1 Functional Interfaces
13.2 Lambda Expressions
13.3 Lambda Expressions and Anonymous Classes
Review Questions
13.4 Overview of Built-In Functional Interfaces

13.5 Suppliers
13.6 Predicates
13.7 Consumers
13.8 Functions
13.9 Two-Arity Specialization of `Function<T, R>`: `BiFunction<T, U, R>`
13.10 Extending `Function<T,T>`: `UnaryOperator<T>`
13.11 Extending `BiFunction<T,T,T>`: `BinaryOperator<T>`
13.12 Currying Functions
13.13 Method and Constructor References
13.14 Contexts for Defining Lambda Expressions
Review Questions

14 Object Comparison

14.1 The `Objects` Class
14.2 Implementing the `equals()` Method
14.3 Implementing the `hashCode()` Method
14.4 Implementing the `java.lang.Comparable<E>` Interface
14.5 Implementing the `java.util.Comparator<E>` Interface
Review Questions

15 Collections: Part II

15.1 The Java Collections Framework
15.2 Collections
15.3 Lists
15.4 Sets
15.5 Sorted Sets and Navigable Sets
15.6 Queues
15.7 Deques
Review Questions
15.8 Maps
15.9 Map Implementations
15.10 Sorted Maps and Navigable Maps

Review Questions

15.11 The `Collections` Class

15.12 The `Arrays` Class

Review Questions

16 Streams

16.1 Introduction to Streams

16.2 Running Example: The `CD` Record Class

16.3 Stream Basics

16.4 Building Streams

16.5 Intermediate Stream Operations

16.6 The `Optional` Class

16.7 Terminal Stream Operations

16.8 Collectors

16.9 Parallel Streams

Review Questions

17 Date and Time

17.1 Date and Time API Overview

17.2 Working with Dates and Times

17.3 Using Temporal Units and Temporal Fields

17.4 Working with Instants

17.5 Working with Periods

17.6 Working with Durations

17.7 Working with Time Zones and Daylight Savings

17.8 Converting Date and Time Values to Legacy Date

Review Questions

18 Localization

18.1 Using Locales

18.2 Properties Files

18.3 Bundling Resources

Review Questions

18.4 Core API for Formatting and Parsing of Values

18.5 Formatting and Parsing Number, Currency, and Percentage Values
18.6 Formatting and Parsing Date and Time
18.7 Formatting and Parsing Messages
Review Questions

19 Java Module System

19.1 Making the Case for Modules
19.2 The Modular JDK
19.3 Module Basics
19.4 Overview of Module Directives
19.5 Creating a Modular Application
19.6 Compiling and Running a Modular Application
19.7 Creating JAR Files
19.8 Open Modules and the `opens` Directive
19.9 Services
19.10 Creating Runtime Images
19.11 Categories of Modules
19.12 Migrating to Modules
19.13 Exploring Modules
19.14 Summary of Selected Operations with the JDK Tools
Review Questions

20 Java I/O: Part I

20.1 Input and Output
20.2 Byte Streams: Input Streams and Output Streams
20.3 Character Streams: Readers and Writers
20.4 The `Console` Class
Review Questions
20.5 Object Serialization
Review Questions

21 Java I/O: Part II

21.1 Characteristics of a Hierarchical File System

21.2 Creating `Path` Objects

21.3 Working with `Path` Objects

21.4 Operations on Directory Entries

21.5 Reading and Writing Files Using Paths

21.6 Managing File Attributes

21.7 Creating Directory Entries

21.8 Stream Operations on Directory Entries

Review Questions

22 Concurrency: Part I

22.1 Threads and Concurrency

22.2 Runtime Organization for Thread Execution

22.3 Creating Threads

Review Questions

22.4 Thread Lifecycle

22.5 Thread Issues

Review Questions

23 Concurrency: Part II

23.1 Utility Classes `TimeUnit` and `ThreadLocalRandom`

23.2 The Executor Framework

23.3 The Fork/Join Framework

23.4 Writing Thread-Safe Code

23.5 Special-Purpose Synchronizers

23.6 Synchronized Collections and Maps

23.7 Concurrent Collections and Maps

Review Questions

24 Database Connectivity

24.1 Introduction to Relational Databases

24.2 Introduction to JDBC

24.3 Establishing a Database Connection

24.4 Creating and Executing SQL Statements

24.5 Processing Query Results

24.6 Customizing Result Sets

- 24.7 Discovering Database and `ResultSet` Metadata
- 24.8 Implementing Transaction Control
- *Review Questions*

25 Annotations

- 25.1 Basics of Annotations
- 25.2 Declaring Annotation Types
- 25.3 Applying Annotations
- 25.4 Meta-Annotations
- 25.5 Selected Standard Annotations
- 25.6 Processing Annotations
- *Review Questions*

26 Secure Coding

- 26.1 Application Security Overview
- 26.2 Security Threat Categories
- 26.3 Java Security Policies
- 26.4 Additional Security Guidelines
- *Review Questions*

A Taking the Java SE 17 and Java SE 11 Developer Exams

- A.1 Preparing for the Exam
- A.2 Registering for the Online Proctored Exam
- A.3 How the Online Proctored Exam Is Conducted
- A.4 The Questions

B Exam Topics: Java SE 17 Developer

C Exam Topics: Java SE 11 Developer

D Annotated Answers to Review Questions

E Mock Exam: Java SE 17 Developer

F Annotated Answers to Mock Exam

G Java Logging API Overview

- G.1 Purpose of the Logging API

G.2 Configuring Logging
G.3 Writing Log Messages
G.4 Applying Guarded Logging
G.5 Summary

Index

Figures

1.1 UML Notation for Classes

1.2 UML Notation for Objects

1.3 Class Diagram Showing Static Members of a Class

1.4 Members of a Class

1.5 Class Diagram Depicting Inheritance Relationship

1.6 Class Diagram Depicting Aggregation

2.1 Primitive Data Types in Java

2.2 Widening Primitive Conversions

2.3 Overflow and Underflow in Floating-Point Arithmetic

2.4 Numeric Promotion in Arithmetic Expressions

3.1 Array of Arrays

3.2 Parameter Passing: Primitive Data Values

3.3 Parameter Passing: Reference Values

3.4 Parameter Passing: Arrays

4.1 Activity Diagram for `if` Statements

4.2 Form of the `switch` Statement with the Colon Notation

4.3 Activity Diagram for the `switch` Statement with the Colon Notation

4.4 Form of the `switch` Statement with the Arrow Notation

4.5 Activity Diagram for the `switch` Statement with the Arrow Notation

4.6 Activity Diagram for the `while` Statement

4.7 Activity Diagram for the `do-while` Statement

4.8 Activity Diagram for the `for` Statement

4.9 Enhanced `for` Statement

5.1 Inheritance Hierarchy

5.2 Inheritance Hierarchy for Example 5.2

5.3 Class Diagram for Example 5.8

5.4 Inheritance Hierarchies for Classes and Interfaces in Example 5.10

5.5 Inheritance Relationships for Interface Constants

5.6 Reference Type Hierarchy: Arrays and Subtype Covariance

5.7 Type Hierarchy to Illustrate Polymorphism

5.8 A Domain Model as Represented by Inheritance Hierarchy

5.9 Sealed Classes for Domain Modeling of Inheritance Hierarchy

5.10 Sealed Classes and Interfaces

6.1 Java Source File Structure

6.2 Package Structure

6.3 Compiling Code into Package Directories

6.4 Searching for Classes

6.5 Accessibility of Class Members

6.6 Block Scope

7.1 Normal Method Execution

7.2 Exception Propagation

7.3 Partial Exception Inheritance Hierarchy

7.4 The `try-catch-finally` Construct

7.5 Exception Handling (Scenario 1)

7.6 Exception Handling (Scenario 2)

7.7 Exception Handling (Scenario 3)

8.1 Partial Inheritance Hierarchy in the `java.lang` Package

8.2 Converting Values among Primitive, Wrapper, and `String` Types

8.3 Incidental Whitespace in Text Blocks

9.1 Overview of Type Declarations
9.2 Static Member Classes and Interfaces
9.3 Outer Object with Associated Inner Objects
9.4 Non-Static Member Classes and Inheritance
9.5 Local Classes and Inheritance Hierarchy
10.1 Memory Organization at Runtime
11.1 Extending Generic Types
11.2 No Subtype Covariance for Parameterized Types
11.3 Partial Type Hierarchy for `Node<? extends Number>`
11.4 Partial Type Hierarchy for `Node<? super Integer>`
11.5 Partial Type Hierarchy for Selected Parameterized Types of `Node<E>` 583
12.1 Partial `ArrayList` Inheritance Hierarchy
15.1 The Core Interfaces
15.2 The Core Collection Interfaces and Their Implementations
15.3 The Core Map Interfaces and Their Implementations
15.4 Bulk Operations on Collections
16.1 Data Processing with Streams
16.2 The Core Stream Interfaces
16.3 Filtering Stream Elements
16.4 Selecting Distinct Elements
16.5 Skipping Elements at the Head of a Stream
16.6 Truncating a Stream
16.7 Mapping
16.8 Incorrect Solution to the Query
16.9 Mapping a Stream of Streams
16.10 Flattening Streams
16.11 Reducing with an Initial Value
16.12 Parallel Functional Reduction
16.13 Sequential Mutable Reduction

16.14 Parallel Mutable Reduction
16.15 Collecting to a Map
16.16 Grouping
16.17 Multilevel Grouping as a Two-Dimensional Matrix
16.18 Partitioning
18.1 Core Classes for Formatting in the `java.text` Package
19.1 Structure of JDK 17 Installation
19.2 Partial Module Graph of Java SE 17 Modules
19.3 Modules and Module Declarations
19.4 Module Graph and Implicit Dependencies
19.5 Module Declarations and Module Graph
19.6 Implied Module Dependencies
19.7 Qualified Export
19.8 Module Graph of the `adviceApp` Application
19.9 Application Directory Structure
19.10 Exploded Module Structure for the `model` Module
19.11 Module Graph and Module Declarations
19.12 Compiling Exploded Modules Containing Source Code
19.13 Compiling the `model` Module
19.14 Creating Modular JARs
19.15 Module Graph and Module Declarations for Services
19.16 Bottom-Up Strategy for Code Migration
19.17 Top-Down Strategy for Code Migration
19.18 Module Graph Using the `jdeps` Tool
20.1 Partial Byte Stream Inheritance Hierarchies in the `java.io` Package
20.2 Stream Chaining for Reading and Writing Binary Values to a File
20.3 Selected Character Streams in the `java.io` Package
20.4 Setting Up a `PrintWriter` to Write to a File
20.5 Setting Up Readers to Read Characters

20.6 Buffered Writers

20.7 Buffered Readers

20.8 Keyboard and Display as Console

20.9 Object Stream Chaining

21.1 Creating `Path` Objects

22.1 Runtime Data Areas

22.2 Spawning Threads Using a `Runnable` Object

22.3 Spawning Threads—Extending the `Thread` Class

22.4 Thread States

22.5 Starting a Thread

22.6 Running and Yielding

22.7 Blocked for Lock Acquisition

22.8 Sleeping and Waking Up

22.9 Timed Waiting and Notifying

22.10 Thread Coordination

22.11 Timed Joining of Threads

22.12 Blocked for I/O

22.13 Thread Termination

22.14 Deadlock

23.1 Executor Interfaces in the `java.util.concurrent` Package

23.2 Executor Service Lifecycle

23.3 Selected Interfaces and Classes in the Lock API

23.4 Concurrent Collections in the `java.util.concurrent` Package

23.5 Concurrent Maps in the `java.util.concurrent` Package

23.6 Blocking Queues in the `java.util.concurrent` Package

23.7 Copy-on-Write Collections in the `java.util.concurrent` Package

24.1 Layered Database Connectivity

25.1 API Documentation of a Deprecated Method

25.2 Selected Types from the Reflection API

Tables

1.1 Terminology for Class Members

1.2 Format Specifier Examples

2.1 Keywords in Java

2.2 Contextual Keywords

2.5 Separators in Java

2.6 Examples of Literals

2.3 Reserved Keywords Not Currently in Use

2.4 Reserved Literals in Java

2.7 Examples of Decimal, Binary, Octal, and Hexadecimal Literals

2.8 Representing Signed `byte` Values Using Two’s Complement

2.9 Examples of Character Literals

2.10 Escape Sequences

2.11 Examples of Escape Sequence `\ddd`

2.13 Range of Character Values

2.14 Range of Floating-Point Values

2.12 Range of Integer Values

2.15 Boolean Values

2.16 Summary of Primitive Data Types

2.17 Selected Conversion Contexts and Conversion Categories

2.18 Operator Summary

2.19 Examples of Truncated Values

2.20 Arithmetic Operators

2.21 Examples of Arithmetic Expression Evaluation

2.22 Arithmetic Compound Assignment Operators

2.23 Relational Operators

2.24 Primitive Data Value Equality Operators

2.25 Reference Equality Operators

2.26 Truth Values for Boolean Logical Operators

2.27 Boolean Logical Compound Assignment Operators

2.28 Conditional Operators

2.29 Truth Values for Conditional Operators

2.30 Integer Bitwise Operators

2.31 Result Table for Bitwise Operators

2.32 Examples of Bitwise Operations

2.33 Bitwise Compound Assignment Operators

2.34 Shift Operators

2.35 Shift Compound Assignment Operators

3.1 Default Values

3.2 Parameter Passing by Value

4.1 Comparing the `switch` Statement and the `switch` Expression

4.2 The `return` Statement

5.1 Overriding, Hiding, and Overloading of Methods

5.2 Summary of Member Declarations in an Interface (Part I)

5.3 Summary of Member Declarations in an Interface (Part II)

5.4 Types and Values

6.1 Compiling and Executing Non-Modular Java Code

6.2 Access Modifiers for Top-Level Reference Types (Non-Modular)

6.3 Accessibility of Members in a Class (Non-Modular)

6.4 Accessing Members within a Class

6.5 Examples of Immutable Classes

7.1 Comparing Checked and Unchecked Exceptions

8.1 Indenting Lines in a String

8.2 Incidental Whitespace in Text Blocks

8.3 Applying Rounding Functions

8.4 Big Number Classes

9.1 Various Aspects of Type Declarations

11.1 Summary of Subtyping Relationships for Generic Types

11.2 Get and Set Operations Using Parameterized References

11.3 Summary of Get and Set Operations Using Parameterized References

11.4 Examples of Type Erasure

11.5 Examples of Reifiable Types

11.6 Examples of Non-Reifiable Types

12.1 Summary of Arrays versus `ArrayList`s

13.1 Basic Functional Interfaces in the `java.util.function` Package

13.2 Built-In Functional Interfaces in the `java.util.function` Package

13.3 Summary of Built-In Functional Interfaces

13.4 Suppliers

13.5 Predicates

13.6 Consumers

13.7 Functions

13.8 Two-arity Functions

13.9 Unary Operators

13.10 Binary Operators

13.11 Method and Constructor References

15.1 Core Interfaces and Concrete Classes in the Collections Framework

15.2 Summary of Collection and Map Implementations

15.3 Bulk Operations and Set Logic

15.4 Summary of Methods in the Queue Interface

15.5 Summary of Deque Methods

15.6 Summary of Scenarios Using Advanced Key-Based Operations

15.7 Map Implementations

16.1 Summary of Stream Building Methods

16.2 Interoperability between Stream Types

16.3 Selected Aspects of Intermediate Stream Operations

16.4 Intermediate Stream Operations

16.5 Terminal Stream Operations

16.6 Additional Terminal Operations in the Numeric Stream Interfaces

16.7 Static Methods in the `Collectors` Class

16.8 Method Comparison: The Stream Interfaces and the `Collectors` Class

17.1 Fields in Selected Classes in the Date and Time API

17.2 Selected Method Name Prefixes in the Temporal Classes

17.3 Selected Common Methods in the Temporal Classes

17.4 Some Differences between the `Period` Class and the `Duration` Class

18.1 Selected Language Codes

18.2 Selected Country/Region Codes

18.3 Selected Predefined Locales for Languages

18.4 Selected Predefined Locales for Countries

18.5 Result and Parent Bundles (See Example 18.3)

18.6 Compact Number Styles

18.7 Formatting Numbers to Compact Form

18.8 Parsing Compact Form to Numbers

18.9 Selected Rounding Modes

18.10 Selected Number Format Pattern Symbols for the United States

18.11 Selected ISO-Based Predefined Formatters for Date and Time

18.13 Using Style-Based Formatters

18.12 Format Styles for Date and Time

18.14 Using Style-Based Parsers

18.15 Selected Date/Time Pattern Letters

18.16 Format Type and Format Style Combinations for Format Elements

19.1 Module Accessibility Rules

19.2 Overview of Module Directives

19.3 The `open` Modifier for Modules

19.4 How JARs Are Handled Depends on the Path

19.5 Module Comparison

19.6 Selected Operations with the JDK Tools

19.7 Selected Options for the `javac` Tool

19.8 Selected Options for the `java` Tool

19.9 Selected Options for the `jar` Tool

19.10 Selected Options for the `jdeps` Tool

19.12 Selected Common Shorthand Options for JDK Tools

19.11 Selected Options for the `jlink` Tool

20.1 Selected Input Streams

20.2 Selected Output Streams

20.3 The `DataInput` and `DataOutput` Interfaces

20.4 Selected Readers

20.5 Selected Writers

20.6 Print Methods of the `PrintWriter` Class

20.7 Correspondence between Selected Byte and Character Streams

21.1 Using Current and Parent Directory Designators

21.2 Converting to an Absolute Path

21.3 Normalizing Paths

21.4 Resolving Paths

21.5 Constructing a Relative Path between Two Paths

21.7 Converting to a Real Path

21.6 Link Option

21.8 Standard Copy Options

21.9 Selected Standard Open Options

21.10 POSIX File Permissions

21.11 Interfaces for Read-Only Access to File Attributes

21.12 Selected File Attribute Views

21.13 File Visit Option

22.1 Thread States Defined by the `Thread.State` Enum Type

22.2 Selective Priorities Defined by the `Thread` Class

23.1 Constants Defined by the `java.util.concurrent.TimeUnit` Enum Type

23.2 Selected Executor Services Provided by the `Executors` Utility Class

23.3 Task-Defining Classes

23.4 Selected Atomic Classes

23.5 Concurrent Collections in the `java.util.concurrent` Package

23.6 Characteristics of Concurrent Collections

23.7 Selected Methods in the `ConcurrentLinkedQueue` Class

23.8 Selected Methods in the `ConcurrentLinkedDeque` Class

23.9 Concurrent Maps in the `java.util.concurrent` Package

23.10 Characteristics of Concurrent Maps

23.11 Blocking Queues in the `java.util.concurrent` Package

23.13 Selected Methods in the `BlockingQueue` Interface

23.12 Characteristics of Blocking Queues

23.14 Selected Methods in the `BlockingDeque` Interface

23.15 Copy-on-Write Collections in the `java.util.concurrent` Package

23.16 Characteristics of Copy-on-Write Collections

24.1 Definition of the `compositions` Table

24.2 Data in the `compositions` Table

24.3 Selected Constants Defined in the `ResultSet` Interface

25.1 Retention Policy Values for the `@Retention` Meta-Annotation

25.2 The `ElementType` Values for the `@Target` Meta-Annotation

25.3 Summary of Selected Standard Annotations

Examples

1.1 Basic Elements of a Class Declaration

1.2 Static Members in Class Declaration

1.3 Defining a Subclass

1.4 Using Aggregation

1.5 A Sample Program

1.6 A Single-File Source-Code Program

2.1 Evaluation Order of Operands and Arguments

2.2 Numeric Promotion in Arithmetic Expressions

2.3 Short-Circuit Evaluation Involving Conditional Operators

2.4 Bitwise Operations

3.1 Default Values for Fields

3.2 Flagging Uninitialized Local Variables of Primitive Data Types

3.3 Flagging Uninitialized Local Reference Variables

3.4 Using the `this` Reference

3.5 Namespaces

3.6 Accessing Static Members in a Class

3.7 Using Arrays

3.8 Using Anonymous Arrays

3.9 Using Multidimensional Arrays

3.10 Passing Primitive Values

3.11 Passing Reference Values

3.12 Passing Arrays

3.13 Array Elements as Primitive Data Values

3.14 Array Elements as Reference Values

3.15 Calling a Variable Arity Method

3.16 Passing Program Arguments

3.17 Illustrating Local Variable Type Reference

4.1 Fall-Through in a `switch` Statement with the Colon Notation

4.2 Nested `switch` Statements with the Colon Notation

4.3 Nested `switch` Statements with the Arrow Notation

4.4 Strings in a `switch` Statement

4.5 Enums in a `switch` Statement

4.6 A `yield` Statement in a `switch` Expression with the Colon Notation

4.7 Statement Blocks in a `switch` Expression with the Arrow Notation

4.8 Expression Actions in a `switch` Expression with the Arrow Notation

4.9 Returning Multiple Values as a Record from a `switch` Expression

4.10 The `break` Statement

4.11 Labeled `break` Statement

4.12 The `continue` Statement

4.13 Labeled `continue` Statement

4.14 The `return` Statement

5.1 Extending Classes: Inheritance and Accessibility

5.2 Overriding, Overloading, and Hiding

5.3 Ambiguous Call to Overloaded Methods

5.4 Using the `super` Keyword

5.5 Constructor Overloading

5.6 The `this()` Constructor Call

5.7 The `super()` Constructor Call

5.8 Using Abstract Classes

5.9 Using the `final` Modifier
5.10 Implementing Interfaces
5.11 Default Methods in Interfaces
5.12 Inheriting Default Method Implementations from Superinterfaces
5.13 Inheriting Method Implementations from Supertypes
5.14 Static Methods in Interfaces
5.15 Private Methods in Interfaces
5.16 Constants in Interfaces
5.17 Inheriting Constants in Interfaces
5.18 Assigning and Passing Reference Values
5.19 Choosing the Most Specific Method (Simple Case)
5.20 Overloaded Method Resolution
5.21 The `instanceof` Type Comparison and Cast Operators
5.22 Using the `instanceof` Type Comparison Operator
5.23 Using Polymorphism
5.24 Using Enums
5.25 Declaring Enum Constructors and Members
5.26 Declaring Constant-Specific Class Bodies
5.27 The `CD` Class
5.28 The `CD` Record Class
5.29 Normal Canonical Record Constructor
5.30 Compact Record Constructor and Non-Canonical Constructor
5.31 Other Member Declarations in a Record Class
5.32 Declaring `sealed`, `non-sealed`, and `final` Classes
5.33 Using Sealed Classes
6.1 Defining Packages and Using Type Import
6.2 Single Static Import
6.3 Avoiding the Interface Constant Antipattern
6.4 Importing Enum Constants

6.5 Shadowing Static Import
6.6 Conflict in Importing a Static Method with the Same Signature
6.7 Access Modifiers for Top-Level Reference Types
6.8 Accessibility of Class Members
6.9 Class Scope
6.10 Implementing an Immutable Class
7.1 Method Execution
7.2 The `try-catch` Construct
7.3 Exception Propagation
7.4 The `try-catch-finally` Construct
7.5 The `try-finally` Construct
7.6 The `finally` Clause and the `return` Statement
7.7 Throwing Exceptions Programmatically
7.8 The `throws` Clause
7.9 Using Multiple `catch` Clauses
7.10 Using the Multi-`catch` Clause
7.11 Precise Rethrowing of Exceptions
7.12 Precise `throws` Clause
7.13 Conditions for Rethrowing Final Exceptions
7.14 Chaining Exceptions
7.15 Naive Resource Management
7.16 Explicit Resource Management
7.17 Using `try` with Automatic Resource Management (ARM)
7.18 Implementing the `AutoCloseable` Interface
8.1 Methods in the `Object` Class
8.2 Text Representation of Integers
8.3 String Construction and Internment
8.4 Reading Characters from a `String`
8.5 Controlling String Builder Capacity

9.1 Overview of Type Declarations

9.2 Static Member Types

9.3 Importing Static Member Types

9.4 Accessing Members in Enclosing Context (Static Member Classes)

9.5 Defining and Instantiating Non-Static Member Classes

9.6 Qualified `this` and Qualified Class Instance Creation Expression

9.7 Inheritance Hierarchy and Enclosing Context

9.8 Extending Inner Classes

9.9 Declaring Local Classes

9.10 Accessing Local Declarations in the Enclosing Block (Local Classes)

9.11 Accessing Members in the Enclosing Class (Local Classes)

9.12 Instantiating Local Classes

9.13 Defining Static Local Record Classes

9.14 Defining Anonymous Classes

9.15 Accessing Declarations in Enclosing Context (Anonymous Classes)

10.1 Initializer Expression Order and Method Calls

10.2 Exceptions in Initializer Expressions

10.3 Static Initializers and Forward References

10.4 Static Initializers and Order of Execution

10.5 Static Initializer Blocks and Exceptions

10.6 Instance Initializers and Forward References

10.7 Instance Initializer Block in Anonymous Class

10.8 Exception Handling in Instance Initializer Blocks

10.9 Object State Construction

10.10 Initialization Anomaly under Object State Construction

11.1 A Legacy Class

11.2 A Generic Class for Nodes

11.3 A Generic Interface and Its Implementation
11.4 A Non-Generic Class Implementing a Generic Interface
11.5 Extending Generic Types
11.6 Unchecked Warnings
11.7 Illustrating Get and Set Operations Using Parameterized References
11.8 Implementing the `Comparable<E>` Interface
11.9 Declaring Generic Methods
11.10 Implementing a Simplified Generic Stack
11.11 Using Recursive Bounds
11.12 Using the `@Override` Annotation
11.13 Subsignatures
11.14 Overriding from Generic Supertype
11.15 Missing Supertype Parameterization
11.16 Genericity Cannot be Added to Inherited Methods
11.17 Restrictions on Exception Handling
11.18 Type Parameter in `throws` Clause
11.19 Generic Nested Types
11.20 Instantiating Generic Nested Classes
11.21 Enum Implements Parameterized Generic Interface
12.1 Using an `ArrayList`
13.1 Accessing Members in the Enclosing Object
13.2 Accessing Local Variables in the Enclosing Method
13.3 Filtering an `ArrayList<E>`
13.4 Implementing Suppliers
13.5 Implementing the `BiPredicate<T, U>` Functional Interface
13.6 Implementing Functions
14.1 Not Overriding the `Object.equals()` Method
14.2 Implications of Not Overriding the `Object.equals()` Method
14.3 Implementing the `equals()` Method

14.4 Implications of Implementing the `equals()` Method

14.5 Implications of Not Overriding the `Object.hashCode()` Method

14.6 Implementing the `hashCode()` Method

14.7 Implications of Implementing the `hashCode()` Method

14.8 Implementing the `compareTo()` Method of the `Comparable<E>` Interface

14.9 Implications of Implementing the `compareTo()` Method

14.10 Natural Ordering and Total Orderings

14.11 Using a Comparator for Version Numbers

15.1 Iterating Over a Collection

15.2 Converting Collections to Arrays

15.3 Using Lists

15.4 Iterating Over Sets

15.5 Using Sets

15.6 Using Navigable Sets

15.7 Using Priority Queues

15.8 Demonstrating Deque Operations

15.9 Using Deques as a LIFO Stack and as a FIFO Queue

15.10 Using Maps

15.11 Using Navigable Maps

16.1 The `CD` Example Classes

16.2 Data Processing Using Streams

16.3 Stream Mapping and Loop Fusion

16.4 Order of Intermediate Operations

16.5 Filtering

16.6 Examining Stream Elements

16.7 Sorting Streams

16.8 Using Optionals

16.9 Using Numerical Optionals

16.10 Identity Matrix Test

16.11 Implementing Functional Reductions

16.12 Implementing Mutable Reductions

16.13 Flat mapping

16.14 Benchmarking

17.1 Creating Local Dates and Local Times

17.2 Using Local Dates and Local Times

17.3 Temporal Arithmetic

17.4 Valid Temporal Units and Temporal Fields

17.5 Period-Based Loop

17.6 More Temporal Arithmetic

17.7 Duration-Based Loop

17.8 Flight Time Information

17.9 Adjusting for DST Crossovers

17.10 Converting to Legacy Date

18.1 Understanding Locales

18.2 The `BasicResources` Bundle Family (See Also Example 18.3)

18.3 Using Resource Bundles (See Also Example 18.2)

18.4 Locating Resource Bundles

18.5 Rounding Modes

18.6 Using the `DecimalFormat` class

18.7 Formatting and Parsing with Letter Patterns

18.8 Formatting Compound Messages

18.9 Using Choice Pattern

18.10 Using the `ChoiceFormat` Class

19.1 Source Code Files for the `adviceApp` Application

19.2 Selected Source Code Files in the `adviceOpen` Application

19.3 Specifying the `Advice` Service Interface

19.4 Implementing the Advice Service Providers

19.5 Implementing the Advice Service Locator

19.6 Implementing of the Advice Service Consumer
20.1 Copying a File Using a Byte Buffer
20.2 Reading and Writing Binary Values
20.3 Copying a File Using a Character Buffer
20.4 Demonstrating Readers and Writers, and Character Encoding
20.5 Changing Passwords
20.6 Object Serialization
20.7 Non-Serializable Objects
20.8 Customized Serialization
20.9 Serialization and Inheritance
20.10 Class Versioning
21.1 Querying `Path` Objects
21.2 Reading and Writing Text Files
21.3 Reading and Writing Bytes
21.4 Accessing Individual Attributes
21.5 Traversing the Directory Hierarchy
22.1 Implementing the `Runnable` Interface
22.2 Extending the `Thread` Class
22.3 Thread States
22.4 Mutual Exclusion
22.5 Interrupt Handling
22.6 Waiting and Notifying
22.7 Joining of Threads
22.8 Thread Termination
22.9 Deadlock
22.10 Livelock
22.11 Starvation
23.1 Executor Lifecycle
23.2 Submitting Tasks and Handling Task Results
23.3 Executing Scheduled One-Shot Tasks

23.4 Executing Scheduled Periodic Tasks
23.5 Task Cancellation
23.6 Filtering Values Using Fork/Join Framework
23.7 Testing Counter Implementations
23.8 Visibility of Shared Data
23.9 Volatile Counter
23.10 Atomic Counter
23.11 Synchronized Counter
23.12 Reentrant Lock Counter
23.13 Reentrant Read-Write Lock Counter
23.14 Cyclic Barrier
23.15 Count-Down Latch
23.16 Serial Access in Synchronized Views of Collections
23.17 Compound Mutually Exclusive Operation
23.18 Concurrent Collections
23.19 Using a Concurrent Map
23.20 Linked Blocking Queue
23.21 Copy-on-Write Array List
24.1 Connecting to the `musicDB` Database
24.2 Executing a Prepared Statement
24.3 Prepared Statement to Execute `INSERT`, `UPDATE`, and `DELETE`
24.4 Stored Procedures and Functions
24.5 Processing a `ResultSet`
24.6 Using `ResultSet` Options
24.7 Discovering Metadata for the Database and `ResultSet`
24.8 Controlling Transactions
25.1 Using the `@Deprecated` Annotation
25.2 Using the `@SuppressWarnings` Annotation
25.3 Using the `@SafeVarargs` Annotation
25.4 Processing Annotations

25.5 Annotation Processor

Foreword

In over 25 years of teaching Java, one of the recurring questions I am asked is why one would bother to pursue Java certification. Obviously one answer is that it might be helpful in pursuit of a job, or a better job, as a Java programmer. But whether that is relevant is dependent on the job market. There is another answer that does not depend on the ever-changing ratio of applicants to vacancies, and it is the answer that I have always preferred. That answer is that it is good for anyone who claims to take a skill seriously to do two things. One is to be measured by the standards of one's peers, and the other is to push oneself to learn and improve continuously. Being satisfied with "good enough" must surely be a sign of some degree of mediocrity.

These days Java training is widely available from many sources (including video-based training that I offer). Each has its pros and cons, but what makes Khalid and Vasily's book stand out is probably the thoroughness of the treatment that is offered in these pages. Khalid has a long history teaching at the university level, and good teaching is a very different skill from merely being an expert. Far too much of what we are offered as training or self-study material these days is, on closer analysis, simply an enumeration of facts. Good teaching requires building not just knowledge but understanding, layer upon layer, so that it sticks.

One aspect of teaching and learning that is sadly little understood is how we learn. If you want to get better at lifting heavy weights, most of us would recognize that somewhere in the process, you have to do just that: lift heavy weights. That is essentially true of all human skills. We must practice (and practice diligently) those things at which we wish to succeed. In the case of learning, what we want to be able to do is to recall facts and apply them on demand. That means, quite literally, that we must practice the act of recalling those facts. To this end, you will find extensive review questions between these covers. These will provide you with the opportunity to practice the act of recall and thereby solidify your knowledge and understanding. That way, not only can you expect to pass what is undoubtedly a challenging exam, but you can also expect to become a better Java programmer.

Good luck in your study!

—Simon Roberts
President, Dancing Cloud Services, LLC

Preface

Writing This Book

Java has been around for over 25 years. During this time, the Java ecosystem has evolved to become the platform of choice for developing software systems, and the Java certification has evolved with it. The goal of this book is twofold: to provide a comprehensive guide, not only for learning Java, and also to nail the Java certification exams.

This book provides extensive coverage for the following Java certifications:

- *Oracle Certified Professional (OCP), Java SE 17 Developer*, and its required exam, *Java SE 17 Developer (1Z0-829)* (for details, see Appendix B)
- *Oracle Certified Professional (OCP), Java SE 11 Developer,* and its required exam, *Java SE 11 Developer (1Z0-819)* (for details, see Appendix C)

The exam syllabus is defined by a set of *exam objectives*. The exam objectives have evolved to become more high-level, with a smattering of specific topics, leaving plenty of room for interpretation as to which topics are covered by an exam objective. The scope of the exam objectives is thus very difficult to define, and entails exhaustive coverage of the Java language features and its many APIs to be on the safe side. This factor is the major cause of the considerable size of this book.

Our approach to writing this book has not changed from the one we employed for our previous books, mainly because it has proved successful. The emphasis remains on analyzing code scenarios, rather than esoteric syntax of individual language constructs. The exam continues to require actual experience with the language, not just mere recitation of facts. We still claim that proficiency in the language is the key to success.

Since the exam emphasizes the core features of Java, this book provides in-depth coverage of topics related to those features.This book is also no different from our previous books in one other important aspect: It is a one-stop guide, providing a mixture of theory and practice that enables readers to prepare for the exam. It can be used to learn Java and to prepare for the exam. After the exam is passed, it can also be used as a handy reference for the Java language.

All elements found in our previous books (e.g., examples, figures, tables, review questions, mock exam questions) can be found in this one as well. We continue to use UML (Unified Modeling Language) extensively to illustrate concepts and language constructs, and all numbered examples continue to be complete Java programs ready for experimentation.

With that, dear reader, we wish you all the best should you decide to go down the path of Java certification. May your loops terminate and your exceptions get caught!

About This Book

This book provides extensive coverage of the core features of the Java programming language and its core application programming interface (API). The book is primarily intended for professionals who want to prepare for the *Java SE 17* or *Java SE 11 Developer Exams*, but it is readily accessible to any programmer who wants to master the language. For both purposes, it provides in-depth coverage of essential features of the language and its core API.

The demand for well-trained and highly skilled Java programmers remains unabated. Oracle offers many Java certifications that professionals can take to validate their skills (see `https://education.oracle.com`). The certification provides members of the IT industry with a standard to use when hiring such professionals, and it allows professionals to turn their Java skills into credentials that are important for career advancement.

The book provides extensive coverage of all the objectives defined by Oracle for both the *Java SE 17* and *Java SE 11 Developer Exams*. The exam objectives are selective, however, and do not include many of the essential features of Java. This book covers many additional topics that every Java programmer should master to be truly proficient. In this regard, the book is a comprehensive primer for learning the Java programming language. After mastering the language by working through this book, the reader can confidently sit for the exams.

This book is *not* a complete reference for Java, as it does not attempt to list every member of every class from the Java SE Platform API documentation. The purpose is not to document the Java SE Platform API. The emphasis is more on the Java programming language features—their syntax and correct usage through code examples— and less on teaching programming techniques.

The book assumes little background in programming. We believe the exam is accessible to any programmer who works through the book. A Java programmer can easily skip over material that is well understood and concentrate on parts that need reinforcing, whereas a programmer new to Java will find the concepts explained from basic principles.

Each topic is explained and discussed thoroughly with examples, and is backed by review questions to reinforce the concepts. The book is not biased toward any particular platform, but provides platform-specific details where necessary.

Using This Book

The reader can choose a linear or a nonlinear route through the book, depending on their programming background. Non-Java programmers wishing to migrate to Java can read Chapter 1, which provides a short introduction to object-oriented programming concepts, and the procedure for compiling and running Java applications. For those preparing for any of the Developer exams, the book has a separate appendix (Appendix A) providing all the pertinent information on preparing for and taking the exams.

Cross-references are provided where necessary to indicate the relationships among the various constructs of the language. To understand a language construct, all pertinent details are provided where the construct is covered, but in addition, cross-references are provided to indicate its relationship to other constructs. Sometimes it is necessary to postpone discussion of certain aspects of a topic if they depend on concepts that have not yet been covered in the book. A typical example is the consequences of object-oriented programming concepts (e.g., inheritance) on the member declarations that can occur in a class. This approach can result in forward references in the initial chapters of the book.

The table of contents; listings of tables, examples, and figures; and a comprehensive index facilitate locating topics discussed in the book.

In particular, we draw attention to the following features of the book:

Chapter Topics

Each chapter starts with a short summary of the topics covered in the chapter, pointing out the major concepts that are introduced.

Developer Exam Objectives

[0.1] Exam objectives that are covered in each chapter are stated clearly at the beginning of every chapter.
[0.2] The number in front of the objective identifies the exam objective, as defined by Oracle. The objectives are organized into major sections, detailing the curriculum for the exam.
[0.3] The objectives for the *Java SE 17* and *Java SE 11 Developer Exams* are reproduced verbatim in Appendix B and Appendix C, respectively. These appendices also map each exam objective to relevant chapters and sections in the book.

Supplementary Topics

- Supplementary topics are Java topics that are *not* on the exam per se, but which the candidate is expected to know.
- Any supplementary topic is listed as a bullet at the beginning of the chapter.

Review Questions

Review questions are provided after every major topic to test and reinforce the material. The review questions predominantly reflect the kind of multiple-choice questions that can be asked on the actual exam. On the exam, the exact number of answers to choose for each question is explicitly stated. The review questions in this book follow that practice.

Many questions on the actual exam contain code snippets with line numbers to indicate that complete implementation is not provided, and that the necessary missing code to compile and run the code snippets can be assumed. The review questions in this book provide complete code implementations where possible, so that the code can be readily compiled and run.

Annotated answers to the review questions are provided in Appendix D.

Example 0.1 *Example Source Code*

We encourage readers to experiment with the code examples to reinforce the material in the book. The source code for the examples can be downloaded from the companion book website (see p. xliii), and readily imported into Eclipse IDE 2022-06.

Java code in the book is presented in `a monospaced font`. Lines of code in the examples or in code snippets are referenced in the text by a number, which is specified by using a single-line comment in the code. For example, in the following code snippet, the call to the method `doSomethingInteresting()` at (1) does something interesting:

Click here to view code image

```
// ...
doSomethingInteresting();                          // (1)
// ...
```

Names of classes and interfaces start with an uppercase letter. Names of packages, variables, and methods start with a lowercase letter. Constants are in all uppercase letters. Interface names begin with the prefix `I`, when it makes sense to distinguish them from class names. Coding conventions are followed, except when we have had to deviate from these conventions in the interest of space or clarity.

Mock Exam

The mock exam for Java SE 17 in Appendix E should be attempted when the reader feels confident about the topics on the exam. It is highly recommended to read Appendix A before attempting the mock exam, as Appendix A contains pertinent information about the questions to expect on the actual exam. Each multiple-choice question in the mock exam explicitly states how many answers are applicable for a given question, as is the case on the actual exam. Annotated answers to the questions in the mock exam are provided in Appendix F.

Java SE Platform API Documentation

> A vertical gray bar is used to highlight methods and fields found in the classes of the Java SE Platform API.
>
> Any explanation following the API information is also similarly highlighted.

To obtain the maximum benefit from using this book in preparing for the *Java SE 17* and *Java SE 11 Developer Exams*, we strongly recommend installing the latest version (Release 17 or newer) of the JDK and its accompanying API documentation. The book focuses solely on Java SE 17 and Java SE 11, and does not acknowledge other versions.

Book Website

This book is backed by a website:

Click here to view code image

```
https://www.mughal.no/jse17ocp/
```

Auxiliary material on the website includes the following:

- Source code for all the examples in the book
- Annotated answers to the reviews questions in the book
- Annotated answers to the mock exam in the book
- Table of contents, sample chapters, and index from the book
- Content specific for the *Java SE 17* and *Java SE 11 Developer Exams*
- Errata for the book
- Links to miscellaneous Java resources (e.g., certification, discussion groups, and tools)

Information about the Java Standard Edition (SE) and its documentation can be found at the following website:

Click here to view code image

```
www.oracle.com/technetwork/java/javase/overview/index.xhtml
```

The current authoritative technical reference for the Java programming language, *The Java® Language Specification: Java SE 17 Edition*, can be found at this website:

Click here to view code image

```
http://docs.oracle.com/javase/specs/index.xhtml
```

Request for Feedback

Considerable effort has been made to ensure the accuracy of the content of this book. All code examples (including code fragments) have been compiled and tested on various platforms. In the final analysis, any errors remaining are the sole responsibility of the principal author.

Any questions, comments, suggestions, and corrections are welcome. Let us know whether the book was helpful (or not) for your purpose. Any feedback is valuable.

The principal author and the co-author can be reached at the following email addresses, respectively:

Click here to view code image

```
khalid@mughal.no
vasiliy.a.strelnikov@oracle.com
```

About the Authors

Khalid A. Mughal

Khalid A. Mughal is the principal author of this book, primarily responsible for writing the material covering the Java topics. He is also the principal author of three other books on previous versions of the Java certification exam: Java SE 8 OCA (1Z0-808), Java SE 6 (1Z0-851), and SCPJ2 1.4 (CX-310-035).

Khalid is an associate professor emeritus at the Department of

Informatics at the University of Bergen, Norway, where he was responsible for designing and implementing various courses in informatics. Over the years, he has taught programming (primarily Java), software engineering (object-oriented system development), databases (data modeling and database management systems), compiler techniques, web application development, and software security courses. For 15 years, he was responsible for developing and running web-based programming courses in Java, which were offered to off-campus students. He has also given numerous courses and seminars at various levels in object-oriented programming and system development using Java and Java-related technologies, both at the University of Bergen and various other universities in Norway and East Africa, and also for the IT industry.

Vasily A. Strelnikov

Vasily Strelnikov is primarily responsible for developing new review questions for the chapters contained in this book.

Vasily is a senior principal OCI (Oracle Cloud Infrastructure) solution specialist, working at Oracle for more than 26 years. He is a co-author of the Java EE 7, Java SE 8, Java SE 11, and Java SE 17 Certification exams. He has designed multiple Java courses that are offered by Oracle: Java SE 17 Programming Complete, Java SE 11 Programming Complete, and Java SE 8 Certification Preparation Seminar. He has also created the Developing Applications for the Java EE 7 Platform training at Oracle. Vasily has over 20 years of experience in Java. He specializes in Java middleware application development and web services.

Acknowledgments

First of all, we would like to acknowledge the contribution that Rolf W. Rasmussen at vizrt made in earlier Java certification books that we wrote together. Some of that contribution still permeates this edition as well.

At Pearson, senior executive editor Greg Doench was once again at the helm. He managed the long process of publishing this book. Senior content producer Julie Nahil was again the in-house contact at Pearson, professionally and efficiently managing production of the book. Audrey Doyle did a truly marvelous job copy-editing the book. She scrutinized every sentence with a meticulous eye—and taught us a thing or two about the English language. The folks at codeMantra did the critical proofreading, endorsing the book with a quality assurance stamp. Rob Mauhar at The CIP Group performed the typesetting wizardry necessary to give the book its professional look. Our sincere thanks to Greg, Julie, Audrey, Rob, the folks at codeMantra, and all those behind the scenes at Pearson, who helped to make this publication see the light of day.

For the technical review of the book, we were lucky to have two Java gurus who agreed to take on the task.

- Mikalai Zaikin is a lead Java developer at IBA Lithuania, and is currently located in Vilnius. He has helped Oracle with development of Java certification exams and has also been a technical reviewer of several Java certification books. He also contributes to the Java Quiz column for Oracle's Java Magazine in collaboration with Simon Roberts.
- Ankit Garg is currently Vice President at Morgan-Stanley, Bengaluru, India, and a Sheriff (i.e., Moderator) for various Java Forums at `coderanch.com`. He is a Java developer with many years' experience in several Java-related technologies (e.g., Java SE, Java EE, and Spring). He is also a certified Java Programmer and Web Component Developer.

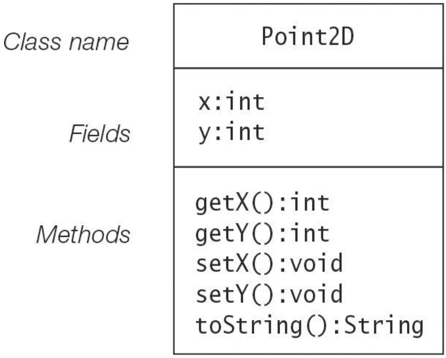
Without doubt, both Mikalai and Ankit have an eye for detail. It is no exaggeration to say that their feedback has been invaluable in improving the quality of this book at all levels. Our most sincere thanks to both Mikalai and Ankit for the many excellent comments and suggestions, and above all, for weeding out numerous pesky errors in the manuscript.

Kristian Berg and Morten Nygaard Åsnes were kind enough to review two chapters each and provide useful feedback, for which we are very grateful.

Great effort has been made to eliminate mistakes and errors in this book. We accept full responsibility for any remaining oversights. We hope that when our diligent readers find any, they will bring them to our attention.

Without family support this edition would still be a fantasy, and we are very grateful to our families for putting up with us when we were burning the midnight oil. Hopefully we can get our family life back now that this book is out the door.

—Khalid A. Mughal
December 2, 2022
Bergen, Norway

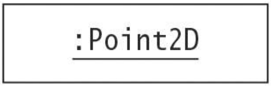

Class name
Point2D
Fields
x:int
y:int
Methods
getX():int
getY():int
setX():void
setY():void
toString():String

(a) Expanded form

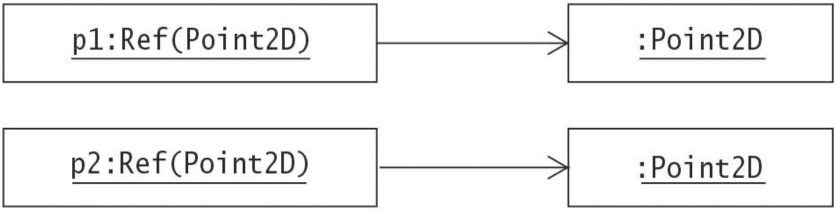
(b) Abbreviated form

p1:Point2D
p2:Point2D

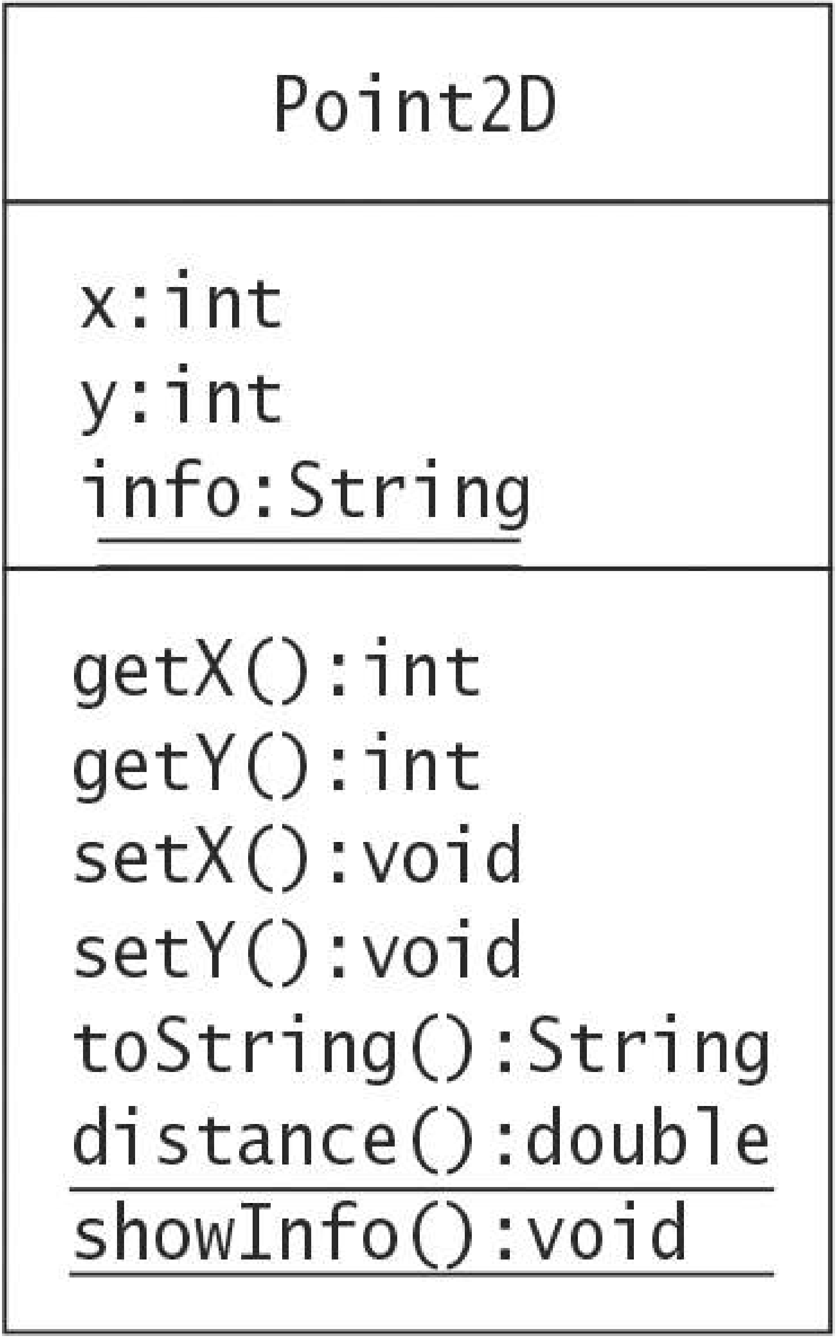
(a) Standard notation for objects

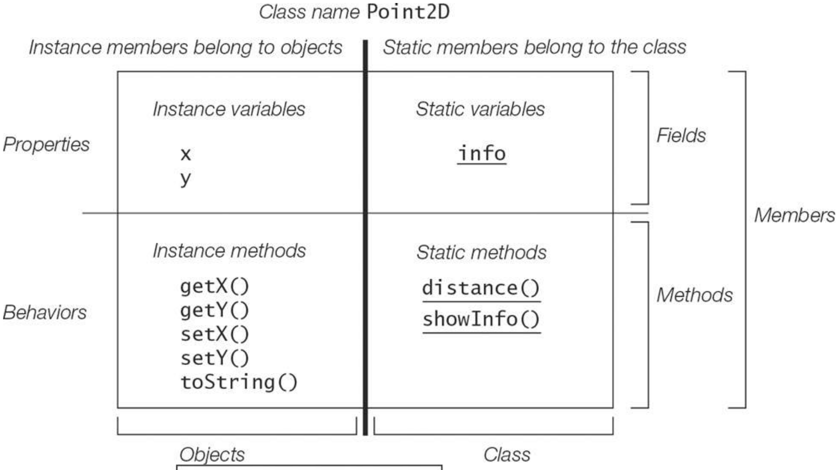


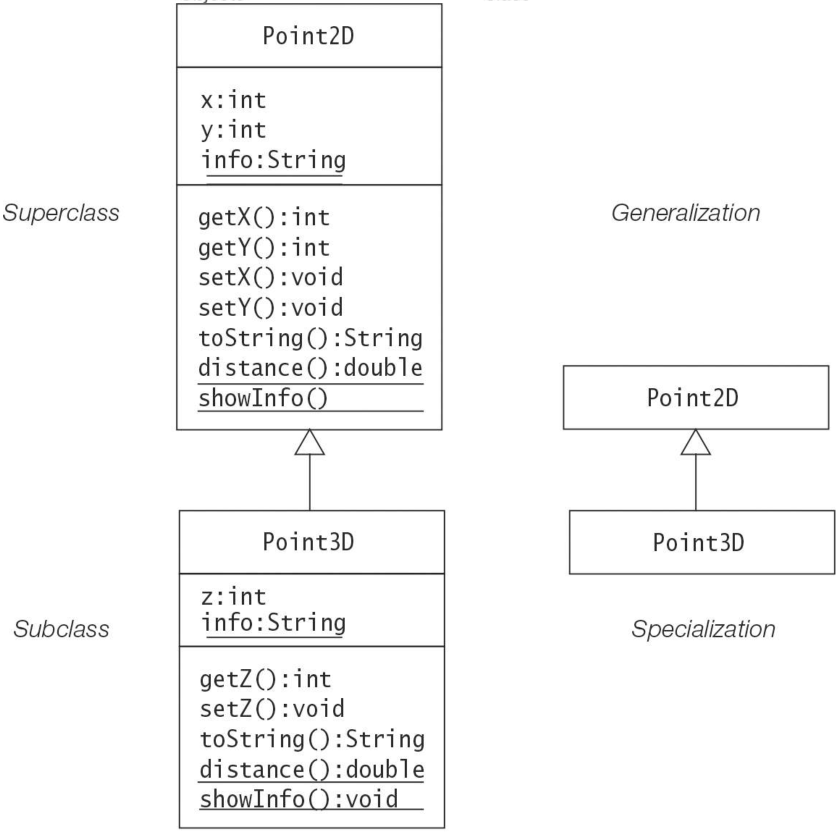
(b) Anonymous object


p1:Ref(Point2D)
:Point2D
p2:Ref(Point2D)
:Point2D

(c) Explicit references for Java objects

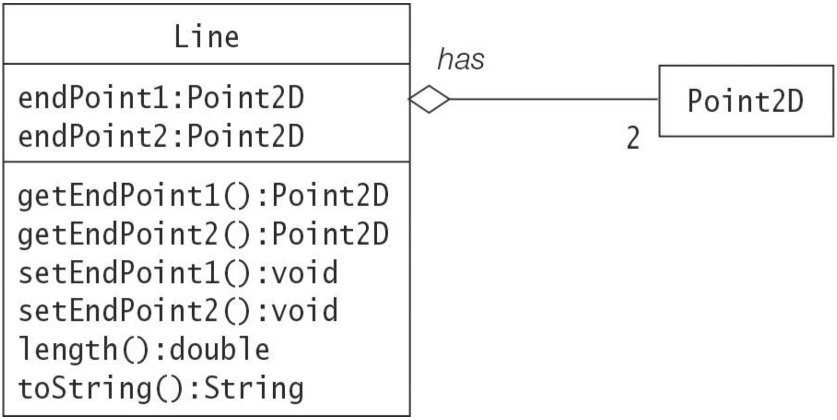

Point2D
x:int
y:int
info:String
getX():int
getY():int
setX():void
setY():void
toString():String
distance():double
showInfo():void


Class name Point2D
Instance members belong to objects
Static members belong to the class
Instance variables
x
y
Static variables
info
Properties
Fields
Members
Instance methods
getX()
getY()
setX()
setY()
toString()
Static methods
distance()
showInfo()
Behaviors
Methods
Objects
Class


Point2D
x:int
y:int
info:String
getX():int
getY():int
setX():void
setY():void
toString():String
distance():double
showInfo()
Superclass
Point3D
z:int
info:String
getZ():int
setZ():void
toString():String
distance():double
showInfo():void
Subclass
Generalization
Point2D
Point3D
Specialization

(a) Expanded form

(b) Abbreviated form


Line
endPoint1:Point2D
endPoint2:Point2D
getEndPoint1():Point2D
getEndPoint2():Point2D
setEndPoint1():void
setEndPoint2():void
length():double
toString():String
has
2
Point2D

Basics of Java Programming

1

Chapter Topics

- Factors and features of the Java ecosystem that have contributed to its evolution and success
- Basic terminology and concepts in Java, and how the language supports object-oriented programming (OOP)
- Understand the distinction between a class and an object
- Basics of how to create objects, access their fields and call methods on them
- Essential elements of a Java program
- Compiling and running Java programs
- Executing single-file source-code programs

- Brief introduction to the Java Shell tool (`jshell`)
- Formatting and printing values to the terminal window

Java SE 17 Developer Exam O	
[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers	§1.2, p. 5 §1.3, p. 8 §1.4, p. 9 §1.5, p. 10
○ *Basic terminology for declaring and using classes and class members is introduced in this chapter.* ○ *For details on classes, fields, methods, and constructors, see Chapter 3, p. 97.* ○ *For record classes, see §5.14, p. 299.* ○ *For instance and static initializers, see §10.5, p. 540.*	

Java SE 11 Developer Exam O	
[3.2] Define and use fields and methods, including instance, static and overloaded methods	§1.2, p. 5 §1.3, p. 8 §1.4, p. 9 §1.5, p. 10
○ *Basic terminology for declaring and using classes and class members is introduced in this chapter.* ○ *For details on classes, fields, methods, and constructors, see Chapter 3, p. 97.*	

Before embarking on the road to Java certification, it is important to understand the basic terminology and concepts in Java. No particular exam objective is covered in this chapter. The emphasis is on providing an introduction to Java and core concepts in object-oriented programming (OOP). In-depth coverage of the concepts introduced will follow in due course in subsequent chapters.

The basic elements of a Java program are introduced in this chapter. The old adage that practice makes perfect is certainly true when learning a programming language. We highly encourage programming on the computer. The mechanics of compiling and running a Java program are provided in this chapter. We begin with an overview of factors that make Java the platform of choice for enterprises and developers.

1.1 The Java Ecosystem

Since its initial release as Java Development Kit 1.0 (JDK 1.0) in 1996, the name Java has become synonymous with a thriving ecosystem that provides the components and the tools necessary for developing systems for today’s multicore world. Its diverse community, comprising a multitude of volunteers, organizations, and corporations, continues to fuel its evolution and grow with its success. Many free and open source technologies now exist that are well proven, mature, and supported, making their adoption less daunting. These tools and frameworks provide support for all phases of the software development lifecycle and beyond.

There are different Java platforms, each targeting different application domains:

- Java SE (Standard Edition): designed for developing desktop and server environments
- Java EE, also known as Jakarta EE (Enterprise Edition): designed for developing enterprise applications
- Java ME (Micro Edition): designed for embedded systems, such as mobile devices and set-top boxes
- Java Card: designed for tiny memory footprint devices, such as smart cards

Each platform provides a hardware/operating system–specific JVM and an API (*application programming interface*) to develop applications for that platform. The Java SE platform provides the core functionality of the language. The Java EE platform is a superset of the Java SE platform and, as the most extensive of the three platforms, targets enterprise application development. The Java ME platform is a subset of the Java SE platform, having a small footprint, and is suitable for developing mobile and embedded applications. The Java Card platform allows development of embedded applications that have a very tiny memory footprint, targeting devices like smart cards. The upshot of this classification is that a Java program developed for one Java platform will not necessarily run under the JVM of another Java platform. The JVM must be compatible with the Java platform that was used to develop the application.

The API and the tools for developing and running Java applications are bundled together as the JDK. Starting with Java 11, JRE (Java Runtime Environment) is no longer available as a stand-alone bundle providing runtime support for execution of Java programs, but it continues to be a subset of the now modular JDK. As before, one needs to install the JDK to both develop and run Java programs. However, to deploy Java programs, the JDK tool `jlink` can be used to create a runtime image that includes the program code and the necessary runtime support to run the program—a topic that we will get to when we discuss modules.

We highly recommend installing the JDK for Java SE 17 depending on the hardware and operating system. Although newer versions of Java are released periodically, Java SE 17 is readily available as an LTS (*long-term support*) release, and is the subject of this book.

As of Java SE 17, Oracle is making the Oracle JDK available for free under the *Oracle No-Fee Terms and Conditions* (NFTC) license. Although subject to the conditions, it permits free use for *all* users.

Key Features of Java

The rest of this section summarizes some of the factors that have contributed to the evolution of Java from an object-oriented programming language to a full-fledged ecosystem for developing all sorts of systems, including large-scale business systems and embedded systems for portable computing devices. A lot of jargon is used in this section and it might be difficult to understand at the first reading, so we recommend coming back after working through the book to appreciate the factors that have contributed to the success of Java.

Multi-paradigm Programming

The Java programming language supports the *object-oriented programming paradigm*, in which the properties of an object and its behavior are encapsulated in the object. The properties and the behavior are represented by the fields and the methods of the object, respectively. The objects communicate through method calls in a procedural manner—in other words, Java also incorporates the *procedural programming paradigm.* Encapsulation ensures that objects are immune to tampering except when manipulated through their public interface. Encapsulation exposes only *what* an object does and not *how* it does it, so that its implementation can be changed with minimal impact on its clients. The later sections in this chapter provide an overview of basic concepts of object-oriented programming, such as inheritance and aggregation, and subsequent chapters will expand on this topic.

Java has also evolved to support the *functional-style programming paradigm* with the introduction of lambda expressions and their implementation using functional interfaces. This topic will be thoroughly explored in this book.

Above all, object-oriented system development promotes code reuse where existing classes can be reused to implement new classes. Its module facility facilitates implementation of large systems, allowing their decomposition into manageable subsystems, as we will see when we discuss modules.

Bytecode Interpreted by the JVM

Java programs are compiled to bytecode that is interpreted by the JVM. Various optimization technologies (e.g., just-in-time [JIT] delivery) have led to the JVM becoming a lean and mean virtual machine with regard to performance, stability, and security. Many other languages, such as Scala, Groovy, and Clojure, now compile to bytecode and seamlessly execute on the JVM. The JVM has thus evolved into an ecosystem in its own right.

Architecture-Neutral and Portable Bytecode

The often-cited slogan "Write once, run everywhere" is true only if a compatible JVM is available for the hardware and software platform. In other words, to run Java SE applications under Windows 10 on a 64-bit hardware architecture, the right JVM must be installed. Fortunately, the JVM has been ported to run under most platforms and operating systems that exist today, including hardware devices such as smart cards, mobile devices, and home appliances.

The specification of the bytecode is architecture neutral, meaning it is independent of any hardware architecture. It is executed by a readily available hardware and operating system–specific JVM. The portability of the Java bytecode thus eases the burden of cross-platform system development.

Simplicity

The language design of Java has been driven by a desire to simplify the programming process. Although Java borrows heavily from the C++ programming language, certain features that were deemed problematic were not incorporated into its design. For example, Java does not have a preprocessor, and it does not allow pointer handling, user-defined operator overloading, or multiple class inheritance.

Java opted for automatic garbage collection, which frees the programmer from dealing with many issues related to memory management, such as memory leaks.

However, the jury is still out on whether the syntax of nested classes or introduction of wild cards for generics can be considered simple.

The introduction of functional-style features has enhanced Java’s appeal, and the potential of its module system is yet to be seen.

Dynamic and Distributed

The JVM can dynamically load class libraries from the local file system as well as from machines on the network, when those libraries are needed at runtime. This feature facilitates linking the code as and when necessary during the execution of a program. It is also possible to programmatically query a class or an object at runtime about its meta-information, such as its methods and fields.

Java provides extensive support for networking to build distributed systems, where objects are able to communicate across networks using various communication protocols and technologies, such as Remote Method Invocation (RMI) and socket connections.

Robust and Secure

Java promotes the development of reliable, robust, and secure systems. It is a strong statically typed language: The compiler guarantees runtime execution if the code compiles without errors. Runtime index checks for arrays and strings, automatic garbage collection, and elimination of pointers are some of the features of Java that promote reliability. The exception handling feature of Java is without a doubt the main factor that facilitates the development of robust systems. And the module system further enhances encapsulation and configuration.

Java provides multilevel protection from malicious code. The language does not allow direct access to memory. A bytecode verifier determines whether any untrusted code loaded in the JVM is safe. The sandbox model is used to confine and execute any untrusted code, limiting the damage that such code can cause. These features, among others, are provided by a comprehensive Java security model to ensure that application code executes securely in the JVM.

High Performance and Multithreaded

The performance of Java programs has improved significantly with various optimizations that are applied to the bytecode at runtime by the JVM. The JIT feature monitors the program at runtime to identify performance-critical bytecode (called *hotspots*) that can be optimized. Such code is usually translated to machine code to boost performance. The performance achieved by the JVM is a balance between native code execution and interpretation of fully scripted languages, which fortunately is adequate for many applications.

Java has always provided high-level support for multithreading, allowing multiple threads of execution to perform different tasks concurrently in an application. It has risen to the new challenges that have emerged in recent years to harness the increased computing power made available by multicore architectures. Functional programming, in which computation is treated as side-effects-free evaluation of functions, is seen as a boon to meet these challenges. Java brings elements of functional-style programming into the language, providing language constructs (lambda expressions and functional interfaces) and API support (through its Concurrent and Stream APIs) to efficiently utilize the many cores to process large amounts of data in parallel.

1.2 Classes

One of the fundamental ways in which we handle complexity is by using *abstractions*. An abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects. The essence of OOP is modeling abstractions, using classes and objects. The hardest part of this endeavor is coming up with the right abstractions.

A *class* denotes a category of objects, and acts as a blueprint for creating objects. A class models an abstraction by defining the properties and behaviors of the objects representing the abstraction. An *object* exhibits the *properties* and *behaviors* defined by its class. The properties of an object of a class are also called *attributes*, and are defined by fields in Java. A *field* in a class is a variable that can store a value that represents a particular property of an object. The behaviors of an object of a class are also known as *operations*, and are defined using *methods* in Java. Fields and methods in a class declaration are collectively called *members*.

An important distinction is made between the *contract* and the *implementation* that a class provides for its objects. The contract defines *which* services are provided, and the implementation defines *how* these services are provided by the class. Clients (i.e., other objects) need only know the contract of an object, and not its implementation, to avail themselves of the object's services.

As an example, we will implement a class that models the abstraction of a point as (x, y)-coordinates in a two-dimensional plane. The class `Point2D` will use two `int` fields `x` and `y` to store the coordinates. Using simplified Unified Modeling Language (UML) notation, the class `Point2D` is graphically depicted in Figure 1.1, which models the abstraction. Both fields, with their type and method names and their return value type, are shown in Figure 1.1a.

Figure 1.1 *UML Notation for Classes*

Declaring Members: Fields and Methods

Example 1.1 shows the declaration of the class `Point2D` depicted in Figure 1.1. Its intention is to illustrate the salient features of a class declaration in Java, rather than an industrial-strength implementation. We will come back to the nitty-gritty of the Java syntax in subsequent chapters.

In Example 1.1, the character sequence `//` in the code indicates the start of a *single-line comment* that can be used to document the code. All characters after this sequence and to the end of the line are ignored by the compiler.

A class declaration can contain member declarations that define the fields and the methods of the objects the class represents. In the case of the class `Point2D`, it has the following two fields declared at (1):

- `x`, which is the x-coordinate of a point
- `y`, which is the y-coordinate of a point

The class `Point2D` has five methods, declared at (3), that implement the essential operations provided by a point:

- `getX()` returns the x-coordinate of the point.
- `getY()` returns the y-coordinate of the point.
- `setX()` sets the x-coordinate to the value passed to the method.

- `setY()` sets the y-coordinate to the value passed to the method.
- `toString()` returns a string with the coordinate values formatted as `"(x,y)"`.

The class declaration also has a method-like declaration at (2) with the same name as the class. Such declarations are called *constructors*. As we shall see, a constructor is executed when an object is created from the class. However, the implementation details in the example are not important for the present discussion.

Example 1.1 *Basic Elements of a Class Declaration*

Click here to view code image

```
// File: Point2D.java
public class Point2D {             // Class name
  // Class Member Declarations

  // Fields:                                                          (1)
  private int x;     // The x-coordinate
  private int y;     // The y-coordinate

  // Constructor:                                                     (2)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Methods:                                                         (3)
  public int  getX()           { return x; }
  public int  getY()           { return y; }
  public void setX(int xCoord) { x = xCoord; }
  public void setY(int yCoord) { y = yCoord; }
  public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y
}
```

1.3 Objects

Class Instantiation, Reference Values, and References

The process of creating objects from a class is called *instantiation*. An *object* is an instance of a class. The object is constructed using the class as a blueprint and is a concrete instance of the abstraction that the class represents. An object must be created before it can be used in a program.

A *reference value* is returned when an object is created. A reference value uniquely denotes a particular object. A *variable* denotes a location in memory where a value can be stored. An *object reference* (or simply *reference*) is a variable that can store a reference value. Thus a reference provides a handle to an object, as it can indirectly denote an object whose reference value it holds. In Java, an object can only be manipulated by a reference that holds its reference value. Direct access to the reference value is not permitted.

This setup for manipulating objects requires that a reference be declared, a class be instantiated to create an object, and the reference value of the object created be stored in the reference. These steps are accomplished by a *declaration statement*:

Click here to view code image

```
Point2D p1 = new Point2D(10, 20);  // A point with coordinates (10,20)
```

In the preceding declaration statement, the left-hand side of the *assignment operator* (`=`) declares that `p1` is a reference of class `Point2D`. The reference `p1`, therefore, can refer to objects of class `Point2D`.

The right-hand side of the assignment operator creates an object of class `Point2D`. This step involves using the `new` operator in conjunction with a call to a constructor of the class (`new Point2D(10, 20)`). The `new` operator creates an instance of the `Point2D` class and returns the reference value of this instance. The arguments passed in the constructor call are used to initialize the `x` and the `y` fields, respectively. The assignment operator stores the reference value in the reference `p1` declared on the left-hand side of the assignment operator. The reference `p1` can now be used to manipulate the object whose reference value it holds.

Analogously, the following declaration statement declares the reference `p2` to be of class `Point2D`, creates an object of class `Point2D`, and assigns its reference value to the reference `p2`:

Click here to view code image

```
Point2D p2 = new Point2D(5, 15);  // A point with coordinates (5,15)
```

Each object that is created has its own copy of the fields declared in the class declaration. That is, the two point objects, referenced by `p1` and `p2`, will have their own `x` and `y` fields. The fields of an object are also called *instance variables*. The values of the instance variables in an object constitute its *state*. Two distinct objects can have the same state if their instance variables have the same values.

The purpose of the constructor call on the right-hand side of the `new` operator is to initialize the fields of the newly created object. In this particular case, for each new `Point2D` object created using the `new` operator, the constructor at (2) in Example 1.1 creates the `x` and `y` fields and initializes them with the arguments passed.

Figure 1.2 shows the UML notation for objects. The graphical representation of an object is very similar to that of a class. Figure 1.2 shows the canonical notation, where the name of the reference denoting the object is prefixed to the class name with a colon (`:`). If the name of the reference is omitted, as in Figure 1.2b, this denotes an anonymous object. Since objects in Java do not have names, but rather are denoted by references, a more elaborate notation is shown in Figure 1.2c, where references of the `Point2D` class explicitly refer to `Point2D` objects. In most cases, the more compact notation will suffice.

Figure 1.2 *UML Notation for Objects*

1.4 Instance Members

The methods of an object define its behavior; such methods are called *instance methods*. It is important to note that these methods pertain to each object of the class. In contrast to the instance variables, the *implementation* of the methods is shared by all instances of the class. Instance variables and instance methods, which belong to objects, are collectively called *instance members*, to distinguish them from *static members* (p. 10), which only belong to the class.

Invoking Methods

Objects communicate by calling methods on each other. As a consequence, an object can be made to exhibit a particular behavior by calling the appropriate method on the object. This is achieved by a *method call* whose basic form is the following: a reference that refers to the object, the binary dot (`.`) operator, and the name of method to be invoked, together with a list of any arguments required by the method.

reference`.`*methodName*`(`*listOfArguments*`)`

The method invoked on the object can also return results back to its caller, via a single return value. The method called must be one that is defined for the object; otherwise, the compiler reports an error.

Click here to view code image

```
Point2D point = new Point2D(-1, -4);   // Creates a point with coordinates (-1
point.setX(-2);                        // (1) The x field is set to the value
int yCoord = point.getY();             // (2) Returns the value -4 of the y fi
System.out.println(point.toString());  // (3) Prints: (-2,-4)
point.distanceFromOrigin();            // (4) Compile-time error: No such meth
```

The sample code above invokes methods on the object denoted by the reference `point`. The method call at (1) sets the value of the `x` field of `point`, and the method call at (2) returns the value of the `y` field of `point`. At (3), the call to the `toString()` method returns the string `"(-2,-4)"` which is printed. The `setX()`, `getY()`, and `toString()` methods are all defined in the class `Point2D`. The `setX()` method does not return any value, but the `getY()` and `toString()` methods do. Trying to invoke a method named `distanceFromOrigin()` at (4) on `point` results in a compile-time error, as no such method is defined in the class `Point2D`.

The dot (`.`) notation can also be used with a reference to access the fields of an object. The basic form for field access is as follows:

reference.*fieldName*

Use of the dot notation is governed by the *accessibility* of the member. The methods of the `Point2D` class are `public` and can thus be called by the clients of the class. However, the fields in the class `Point2D` have `private` access, indicating that they are not accessible from outside the class. Thus the code below at (1) in a client of the `Point2D` class will not compile. Typically, a class provides `public` methods to access values in its `private` fields, as class `Point2D` does.

Click here to view code image

```
System.out.println(point.x);       // (1) Compile-time error: x is not accessi
System.out.println(point.getX());  // OK.
```

1.5 Static Members

In some cases, certain members should belong only to the class; that is, they should not be part of any instance of the class. Such members are called *static members*. Fields and methods that are static members are easily distinguishable in a class declaration as they must always be declared with the keyword `static`.

Figure 1.3 shows the class diagram for the class `Point2D`. It has been augmented by three static members, whose names are underlined to distinguish them from instance members. The augmented declaration of the `Point2D` class is given in Example 1.2.

Figure 1.3 *Class Diagram Showing Static Members of a Class*

In Example 1.2, the field `info` at (1) is declared as a *static variable*. This field has information about the purpose of the class that the class can share with its clients. A static variable belongs to the class, rather than to any specific object of the class. It will be allocated in the class and initialized to the string specified in its declaration when the class is loaded. Declaring the `info` field as `static` makes sense, as it is unnecessary that every object of the class `Point2D` should have a copy of this information.

Click here to view code image

```
private static String info = "A point represented by (x,y)-coordinates.";
```

In Example 1.2, the two methods `distance()` and `showInfo()` at (5) are *static methods* belonging to the class. Both are declared with the keyword `static`. The static method `distance()` calculates and returns the distance between two points passed as arguments to the method. The static method `showInfo()` prints the string with the information referenced by the static variable `info`. These methods belong to the class, rather than to any specific objects of the class.

Clients can access static members in the class by using the class name. The following code invokes the static method `distance()` in the class `Point2D`:

Click here to view code image

```
double d = Point2D.distance(p1, p2); // Class name to invoke static method
```

Static members can also be accessed via object references, although doing so is not encouraged:

Click here to view code image

```
p1.showInfo();                       // Reference invokes static method
```

Static members in a class can be accessed both by the class name and via object references, but instance members can be accessed only by object references.

Example 1.2 *Static Members in Class Declaration*

Click here to view code image

```
// File: Point2D.java
```

```
public class Point2D {              // Class name
  // Class Member Declarations
  // Static variable:                                                   (1)
  private static String info = "A 2D point represented by (x,y)-coordinates.";

  // Instance variables:                                                (2)
  private int x;
  private int y;

  // Constructor:                                                       (3)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Instance methods:                                                  (4)
  public int  getX()            { return x; }
  public int  getY()            { return y; }
  public void setX(int xCoord) { x = xCoord; }
  public void setY(int yCoord) { y = yCoord; }
  public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y

  // Static methods:                                                    (5)
  public static double distance(Point2D p1, Point2D p2) {
    int xDiff = p1.x - p2.x;
    int yDiff = p1.y - p2.y;
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
  }
  public static void showInfo() { System.out.println(info); }
}
```

Figure 1.4 shows the classification of the members in the class `Point2D`, using the terminology we have introduced so far. Table 1.1 provides a summary of the terminology used in defining members of a class.

Figure 1.4 *Members of a Class*

Table 1.1 *Terminology for Class Members*

Instance members	The instance variables and instance methods of an object. They can be accessed or invoked only through an object reference.
Instance variable	A field that is allocated when the class is instantiated (i.e., when an object of the class is created). Also called a *non-static field* or just a *field* when the context is obvious.
Instance method	A method that belongs to an instance of the class. Objects of the same class share its implementation.
Static members	The static variables and static methods of a class. They can be accessed or invoked either by using the class name or through an object reference.
Static variable	A field that is allocated when the class is loaded. It belongs to the class, and not to any specific object of the class. Also called a *static field* or a *class variable*.
Static method	A method that belongs to the class, and not to any object of the class. Also called a *class method*.

1.6 Inheritance

There are two fundamental mechanisms for building new classes from existing ones: *inheritance* and *aggregation* (p. 16). It makes sense to *inherit* from an existing class `Vehicle` to define a class `Car`, since a car is a vehicle. The class `Vehicle` has several *parts*; therefore, it makes sense to define a *composite object* of the class `Vehicle` that has *constituent objects* of such classes as `Engine`, `Axle`, and `GearBox`, which make up a vehicle.

Inheritance is illustrated here by an example that implements a point in three-dimensional space—that is, a 3D point represented by (x, y, z)-coordinates. We can derive the 3D point from the `Point2D` class. This 3D point will have all the properties and behaviors of the `Point2D` class, along with the additional third dimension. This relationship is shown in Figure 1.5 and implemented in Example 1.3. The class `Point3D` is called the *subclass*, and the class `Point2D` is called the *superclass*. The `Point2D` class is a *generalization* for points, whereas the class `Point3D` is a *specialization* of points that have three coordinates.

Figure 1.5 *Class Diagram Depicting Inheritance Relationship*

In Java, deriving a new class from an existing class requires the use of the `extends` clause in the subclass declaration. A subclass can *extend* only one superclass. The subclass `Point3D` extends the `Point2D` class, shown at (1).

Click here to view code image

```
public class Point3D extends Point2D {                // (1) Uses extends clause
  // ...
}
```

The `Point3D` class only declares the z-coordinate at (3), as every object of the subclass will have the `x` and `y` fields that are specified in its superclass `Point2D`. Note that these fields are declared `private` in the superclass `Point2D`, but they are accessible to a `Point3D` object indirectly through the `public` get and set methods for the x- and the y-coordinates in the superclass `Point2D`. These methods are *inherited* by the `Point3D` class.

The constructor of the `Point3D` class at (4) takes three arguments corresponding to the x-, y-, and z-coordinates. The call to `super()` at (5) results in the constructor of the superclass `Point2D` being called to initialize the x- and `y`-coordinates.

It addition, the `Point3D` class declares methods at (6) to get and set the z-coordinate. It provides its own version of the `toString()` method to format a point that has three coordinates.

Since calculating the distance is also different in three-dimensional space from that in a two-dimensional plane, the `Point3D` class provides its own `distance()` static method at (7). As its objects represent 3D points, it declares its own static field `info` and provides its own static method `showInfo()` to print this information.

Example 1.3 *Defining a Subclass*

Click here to view code image

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

Click here to view code image

```
// File: Point3D.java
public class Point3D extends Point2D {            // (1) Uses extends clause

  // Static variable:                                                    (2)
  private static String info = "A 3D point represented by (x,y,z)-coordinates.

  // Instance variable:                                                  (3)
  private int z;

  // Constructor:                                                        (4)
  public Point3D(int xCoord, int yCoord, int zCoord) {
    super(xCoord, yCoord);                                            // (5)
    z = zCoord;
  }

  // Instance methods:                                                   (6)
  public int  getZ()            { return z; }
  public void setZ(int zCoord)  { z = zCoord; }
  @Override
  public String toString() {
    return "(" + getX() + "," + getY() + "," + z + ")"; // Format: (x,y,z)
  }

  // Static methods:                                                     (7)
  public static double distance(Point3D p1, Point3D p2) {
    int xDiff = p1.getX() - p2.getX();
    int yDiff = p1.getY() - p2.getY();
    int zDiff = p1.getZ() - p2.getZ();
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff + zDiff*zDiff);
  }
  public static void showInfo() { System.out.println(info); }
}
```

Objects of the `Point3D` class will respond just like objects of the `Point2D` class, but they also have the additional functionality defined in the subclass. References of the class `Point3D` are used in the code below. The comments indicate in which class a method is invoked. Note that the subclass reference can invoke the inherited get and set methods in the superclass.

Click here to view code image

```
Point3D p3A = new Point3D(10, 20, 30);
System.out.println(p3A.toString());               // (10,20,30)       (Point3D
System.out.println("x: " + p3A.getX());           // x: 10            (Point2D
System.out.println("y: " + p3A.getY());           // y: 20            (Point2D
System.out.println("z: " + p3A.getZ());           // z: 30            (Point3D

p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
System.out.println(p3A.toString());               // (-10,-20,-30)    (Point3D

Point3D p3B = new Point3D(30, 20, 10);
System.out.println(p3B.toString());               // (30,20,10)       (Point3D
System.out.println(Point3D.distance(p3A, p3B));   // 69.2820323027551 (Point3D
Point3D.showInfo(); // A 3D point represented by (x,y,z)-coordinates. (Point3D
```

1.7 Aggregation

An *association* defines a static relationship between objects of two classes. One such association, called *aggregation* (also known as *composition*), expresses how an object uses other objects. Java supports aggregation of objects by reference, since objects cannot contain other objects explicitly. The aggregate object usually has fields that denote its constituent objects. By default, Java uses aggregation when fields denoting objects are declared in a class declaration. Typically, an aggregate object delegates its tasks to its constituent objects.

We illustrate aggregation by implementing a finite-length straight line that has two end points in a two-dimensional plane. We would like to use the class `Point2D` to implement such a line. A class `Line` could be implemented by having fields for two `Point2D` objects that would represent the end points of a line. This aggregate relationship is depicted in Figure 1.6, which shows that a `Line` object has two `Point2D` objects, indicated by the diamond notation. The complete declaration of the `Line` class is shown in Example 1.4. The two fields `endPoint1` and `endPoint2` declared at (1) represent the two end points. In particular, note the `length()` method at (2) which delegates the computation of the length to the `Point2D.distance()` method.

Figure 1.6 *Class Diagram Depicting Aggregation*

The following code shows how a `Line` object can be manipulated:

Click here to view code image

```
Line line1 = new Line(new Point2D(5,6), new Point2D(7,8));
System.out.println(line1.toString());               // Line[(5,6),(7,8)]
line1.setEndPoint1(new Point2D(11, 12));
line1.setEndPoint2(new Point2D(13, 14));
System.out.println(line1.toString());               // Line[(11,12),(13,14)]
System.out.println("Length: " + line1.length());    // Length: 2.8284271247461
```

Example 1.4 *Using Aggregation*

Click here to view code image

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}


// File: Line.java
public class Line {

  // Instance variables:                                                    (1
  private Point2D endPoint1;
  private Point2D endPoint2;

  // Constructor:
  public Line(Point2D p1, Point2D p2) {
    endPoint1 = p1;
    endPoint2 = p2;
  }

  // Methods:
  public Point2D getEndPoint1() { return endPoint1; }
  public Point2D getEndPoint2() { return endPoint2; }
  public void setEndPoint1(Point2D p1) { endPoint1 = p1; }
  public void setEndPoint2(Point2D p2) { endPoint2 = p2; }
  public double length() {                                               // (2
    return Point2D.distance(endPoint1, endPoint2);
  }
  public String toString()  {
    return "Line[" + endPoint1 + "," + endPoint2 + "]";
  }
}
```

Review Questions

1.1 Which statement is true about methods?

Select the one correct answer.

a. A method is an attribute defining a particular property of an abstraction.

b. A method is a category of objects.

c. A method is an operation defining a particular behavior of an abstraction.

d. A method is a blueprint for defining operations.

1.2 Which statement is true about objects?

Select the one correct answer.

a. An object is what classes are instantiated from.

b. An object is an instance of a class.

c. An object is a blueprint for creating concrete realization of abstractions.

d. An object is a reference.

e. An object is a variable.

1.3 Which is the first line of a constructor declaration in the following code?

Click here to view code image

```
public class Counter {                                             // (1
  int current, step;
  public Counter(int startValue, int stepValue) {                  // (2
    setCurrent(startValue);                                        // (3
    setStep(stepValue);
  }
  public int  getCurrent()           { return current; }           // (4
  public void setCurrent(int value)  { current = value; }          // (5
  public void setStep(int stepValue) { step = stepValue; }         // (6
}
```

Select the one correct answer.

a. (1)

b. (2)

c. (3)

d. (4)

e. (5)

f. (6)

1.4 Given that `Thing` is a class, how many objects are created and how many references are declared by the following code?

Click here to view code image

```
Thing item, stuff;
item = new Thing();
Thing entity = new Thing();
```

Select the two correct answers.

a. One object is created.

b. Two objects are created.

c. Three objects are created.

d. One reference is declared.

e. Two references are declared.

f. Three references are declared.

1.5 Which statement is true about instance members?

Select the one correct answer.

a. An instance member is also called a static member.
b. An instance member is always a field.
c. An instance member is never a method.
d. An instance member is always a part of an instance.
e. An instance member always represents an operation.

1.6 How do objects communicate with each other in Java?

Select the one correct answer.

a. They communicate by modifying each other's fields.
b. They communicate by modifying the static variables of each other's classes.
c. They communicate by calling each other's instance methods.
d. They communicate by calling static methods of each other's classes.

1.7 Given the following code, which of the following statements are true?

Click here to view code image

```
class A {
  protected int value1;
}

class B extends A {
  int value2;
}
```

Select the two correct answers.

a. Class `A` extends class `B`.
b. Class `B` is the superclass of class `A`.
c. Class `A` inherits from class `B`.
d. Class `B` is a subclass of class `A`.
e. Objects of class `A` have a field named `value2`.
f. Objects of class `B` have a field named `value1`.

1.8 Sample Java Program

The term *program* refers to source code that is compiled and directly executed. The terms *program* and *application* are often used synonymously, and are so used in this book. To create a program in Java, the program must have a class that defines a method named `main`, which is invoked at runtime to start the execution of the program. The class with this `main()` method is known as the *entry point of the program*.

Essential Elements of a Java Program

Example 1.5 comprises three classes: `Point2D`, `Point3D`, and `TestPoint3D`. The public class `TestPoint3D` in the file `TestPoint3D.java` is the entry point of the program. It defines a method with the name `main`. The *method header* of this `main()` method must be declared as shown in the following method stub:

Click here to view code image

```
public static void main(String[] args)    // Method header
{ /* Implementation */ }
```

The `main()` method has `public` access—that is, it is accessible from any class. The keyword `static` means the method belongs to the class. The keyword `void` indicates that the method does not return any value. The parameter `args` is an array of strings that can be used to pass information to the `main()` method when execution starts.

Example 1.5 *A Sample Program*

Click here to view code image

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

Click here to view code image

```
// File: Point3D.java
public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

Click here to view code image

```
// File: TestPoint3D.java
public class TestPoint3D {
  public static void main(String[] args) {
    Point3D p3A = new Point3D(10, 20, 30);
    System.out.println("p3A: " + p3A.toString());
    System.out.println("x: " + p3A.getX());
    System.out.println("y: " + p3A.getY());
    System.out.println("z: " + p3A.getZ());
    p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
    System.out.println("p3A: " + p3A.toString());
    Point3D p3B = new Point3D(30, 20, 10);
    System.out.println("p3B: " + p3B.toString());
    System.out.println("Distance between p3A and p3B: " +
                       Point3D.distance(p3A, p3B));
    Point3D.showInfo();
  }
}
```

Output from the program:

Click here to view code image

```
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

Compiling a Program

The JDK provides tools for compiling and running programs. The classes in the Java SE Platform API are already compiled, and the JDK tools know where to find them.

Java source files can be compiled using the *Java Language Compiler*, `javac`, which is part of the JDK. Each source file name has the extension `.java`. Each class declaration in a source file is compiled into a separate *class file*, containing its *Java bytecode*. The name of this file comprises the name of the class with `.class` as its extension.

The source files `Point2D.java`, `Point3D.java`, and `TestPoint3D.java` contain the declarations of the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. The respective source files are in the same directory. The source files can be compiled by giving the following `javac` command on the command line (the character `>` is the command prompt and we will use bold type for anything typed on the command line):

Click here to view code image

```
>javac Point2D.java Point3D.java TestPoint3D.java
```

This `javac` command creates the class files `Point2D.class`, `Point3D.class`, and `TestPoint3D.class` containing the Java bytecode for the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. The command creates the class files in the same directory as the source files.

Although a Java source file can contain more than one class declaration, the Java compiler enforces the rule that there can only be *at the most* one class in the source file that has `public` access. If there is a public class in the source file, the name of the source file must be the name of the public class with `.java` as its extension. In the absence of a public class in the source file, the name of the file can be arbitrary, but still with the `.java` extension. Regardless, each class declaration in a source file is compiled into a separate `.class` file.

Running a Program

It is the bytecode in the class files that is executed when a Java program is run—the source code is immaterial in this regard. A Java program is run by the *Java Application Launcher*, `java`, which is also part of the JDK. The `java` command creates an instance of the JVM that executes the bytecode.

The following `java` command on the command line will launch the program in Example 1.5:

```
>java TestPoint3D
p3A: (10,20,30)
...
```

Note that only the name of the class that is the entry point of the program is specified, resulting in the execution starting in the `main()` method of the specified class. This `main()` method is found in the *class* file of the `TestPoint3D` class. The program in Example 1.5 terminates when the execution of the `main()` method is completed.

Running a Single-File Source-Code Program

Typically, Java source code is first compiled by the `javac` command to Java byte-code in class files and then the bytecode in the class files is executed by the `java` command. The compilation step can be omitted if the complete source code of the program is contained in a *single* source file, meaning that all class declarations that comprise the program are declared in one source file.

In Example 1.5, the program is composed of three source files: `Point2D.java`, `Point3D.java`, and `TestPoint3D.java`, containing the declarations of the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. In Example 1.6, the class declarations are now contained in the `Demo-App.java` file; in other words, the complete source code of the program is now in a single source file. We can run the program with the following `java` command, without compiling the source code first:

```
>java Demo-App.java
p3A: (10,20,30)
...
```

The full name of the single source file is specified in the command line. Full program output is shown in Example 1.6.

Note that no class files are created. The source code is compiled fully in memory and executed.

In order to run a single-file source-code program, the following conditions must be met:

- The single source file must contain *all* source code for the program.
- Although there can be several class declarations in the source file, the *first* class declaration in the source file must provide the `main()` method; that is, it must be the entry point of the program.
- There must not exist *class* files corresponding to the class declarations in the single source file that are accessible by the `java` command.

Unlike the `javac` command, the name of the single source file (e.g., `Demo-App.java`) need not be a valid class name, but it must have the `.java` extension. Also unlike the `javac` command, the `java` command allows several public classes in the single source file (only public classes in the `Demo-App.java` file).

Examples of single-file source-code programs can be found throughout the book.

Example 1.6 *A Single-File Source-Code Program*

Click here to view code image

```
// File: Demo-App.java
public class TestPoint3D {
  // Same as in Example 1.5.
  // Provides the main() method and is the first class declaration in the file
}

public class Point2D {
  // Same as in Example 1.2.
}

public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

Running the program:

Click here to view code image

```
>java Demo-App.java
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

The Java Shell Tool (`jshell`)

This subsection is not on any Java Developer Exam. Its sole purpose is to introduce a JDK tool that is an excellent aid in learning Java programming.

The interactive command-line tool `jshell` is excellent when it comes to learning the Java programming language. It is a *Read-Evaluate-Print Loop* (REPL) tool, meaning that it continuously reads what is typed at the terminal, evaluates the input, and prints the results. It evaluates such language constructs as declarations, statements, and expressions as they are entered at the terminal, and shows the results immediately. It provides access to the Java SE Platform API. Pressing the TAB key results in auto-completion of the snippet, and if that fails, suggests possible options. It is an ideal tool for quickly testing code snippets. We also encourage the reader to consult the documentation for the `jshell` JDK tool.

The following is an example of a session with the `jshell` tool:

Click here to view code image

```
>jshell
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> int i = 20
i ==> 20

jshell> Math.sqrt(i)
$6 ==> 4.47213595499958

jshell> 3 + 4 * 5
$7 ==> 23
```

```
jshell> (3 + 4) * 5
$8 ==> 35

jshell> /exit
|  Goodbye
>
```

1.9 Program Output

Data produced by a program is called *output*. This output can be sent to different devices. The examples presented in this book usually send their output to a terminal window, where the output is printed as a line of characters with a cursor that advances as the characters are printed. A Java program can send its output to the terminal window using an object called *standard out*. This object, which can be accessed using the public static final field `out` in the `System` class, is an object of the class `java.io.PrintStream`. This class provides methods for printing values. These methods convert values to their text representation and print the resulting string.

The print methods convert a primitive value to a string that represents its literal value, and then print the resulting string.

Click here to view code image

```
System.out.println(2022);                                  // 2022
```

An object is first converted to its text representation by calling its `toString()` method implicitly, if it is not already called explicitly on the object. The print statements below will print the same text representation of the `Point2D` object denoted by the reference `origin`:

Click here to view code image

```
Point2D origin = new Point2D(0, 0);
System.out.println(origin.toString());                     // (0,0)
System.out.println(origin);                                // (0,0)
```

The `toString()` method called on a `String` object returns the `String` object itself. As string literals are `String` objects, the following statements will print the same result:

Click here to view code image

```
System.out.println("Stranger Strings".toString());         // Stranger Strings
System.out.println("Stranger Strings");                    // Stranger Strings
```

The `println()` method always terminates the current line, which results in the cursor being moved to the beginning of the next line. The `print()` method prints its argument to the terminal window, but it does not terminate the current line:

Click here to view code image

```
System.out.print("Don't terminate this line!");
```

To terminate a line without printing any values, we can use the no-argument `println()` method:

```
System.out.println();
```

Formatted Output

This subsection is not on any Java Developer Exam. It is solely included because many examples in this book format their output to aid in understanding the computed results.

For more control over how the values are printed, we can format the output. The following method of the `java.io.PrintStream` class can be used for this purpose:

> Click here to view code image
>
> ```
> PrintStream printf(String format, Object... args)
> ```
>
> The `String` parameter `format` specifies how formatting will be done. It contains *format specifications* that determine how each subsequent value in the parameter `args` will be formatted and printed. The parameter declaration `Object... args` represents an array of zero or more arguments to be formatted and printed. The resulting string from the formatting will be printed to the *destination stream*. (`System.out` will print to the *standard out* object.)
>
> Any error in the format string will result in a runtime exception.
>
> This method returns the `PrintStream` on which the method is invoked, and can be ignored, as in the examples here.

The following call to the `printf()` method on the standard `out` object formats and prints three values:

Click here to view code image

```
System.out.printf("Formatted values|%5d|%8.3f|%5s|%n", // Format string
                   2016, Math.PI, "Hi");               // Values to format
```

At runtime, the following line is printed in the terminal window:

Click here to view code image

```
Formatted values|  2016|   3.142|    Hi|
```

The format string is the first argument in the method call. It contains four *format specifiers*. The first three are `%5d`, `%8.3f`, and `%5s`, which specify how the three arguments should be processed. The letter in the format specifier indicates the type of value to format. Their location in the format string specifies where the text representation of the arguments should be inserted. The fourth format specifier, `%n`, is a platform-specific line separator. Its occurrence causes the current line to be terminated, with the cursor moving to the start of the next line. All other text in the format string is fixed, including any other spaces or punctuation, and is printed verbatim.

In the preceding example, the first value is formatted according to the first format specifier, the second value is formatted according to the second format specifier, and so on. The | character has been used in the format string to show how many character positions are taken up by the text representation of each value. The output shows that the `int` value was written right-justified, spanning five character positions using the format specifier `%5d`; the `double` value of `Math.PI` took up eight character positions and was rounded to three decimal places using the format specifier `%8.3f`; and the `String` value was written right-justified, spanning five character positions using the format specifier `%5s`. The format specifier `%n` terminates the current line. All other characters in the format string are printed verbatim.

Table 1.2 shows examples of some selected format specifiers that can be used to format values.

Table 1.2 *Format Specifier Examples*

Parameter value	Format spec	Example value	String printed	Description
Integer value	"%d"	123	"123"	Occupies as many character positions as needed.
	"%6d"	123	" 123"	Occupies six character positions and is right-justified. The printed string is padded with leading spaces, if necessary.
Floating-point value	"%f"	4.567	"4.567000"	Occupies as many character positions as needed, but always includes six decimal places.
	"%.2f"	4.567	"4.57"	Occupies as many character positions as needed, but includes only two decimal places. The value is rounded in the output, if necessary.
	"%6.2f"	4.567	" 4.57"	Occupies six character positions, including the decimal point, and

uses two decimal places. The value is rounded in the output, if necessary.

Any object [illegible] The text representation of the object occupies as many character positions as needed.

[illegible] The text representation of the object occupies six character positions and is right-justified.

[illegible] The text representation of the object occupies six character positions and is left-justified.

Review Questions

1.8 Which command from the JDK will create a class file with the bytecode of the following source code contained in a file named `SmallProg.java`?

Click here to view code image

```
public class SmallProg {
  public static void main(String[] args) { System.out.println("Good luck
}
```

Select the one correct answer.

- **a.** `java SmallProg`
- **b.** `javac SmallProg`
- **c.** `java SmallProg.java`
- **d.** `javac SmallProg.java`
- **e.** `java SmallProg main`

1.9 Which command from the JDK should be used to execute the `main()` method of a class named `SmallProg` that has been compiled?

Select the one correct answer.

- **a.** `java SmallProg`
- **b.** `javac SmallProg`
- **c.** `java SmallProg.java`
- **d.** `java SmallProg.class`
- **e.** `java SmallProg.main()`

1.10 Which of the following statements are true about a single-file source-code program?

Select the two correct answers.

a. It can be composed of multiple class declarations in the source file, where the first class declaration must provide the `main()` method.

b. It can access previously compiled user-defined classes.

c. It can be composed of multiple source files.

d. It can accept program arguments on the command line.

1.11 Which statement is true about Java?

Select the one correct answer.

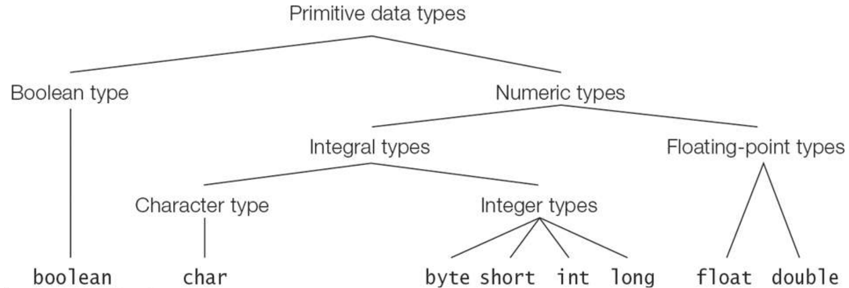
a. A Java program can be executed by any JVM.

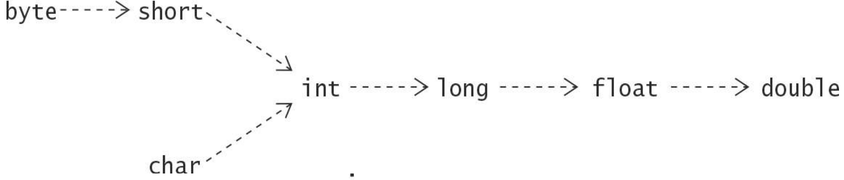
b. Java bytecode cannot be translated to machine code.

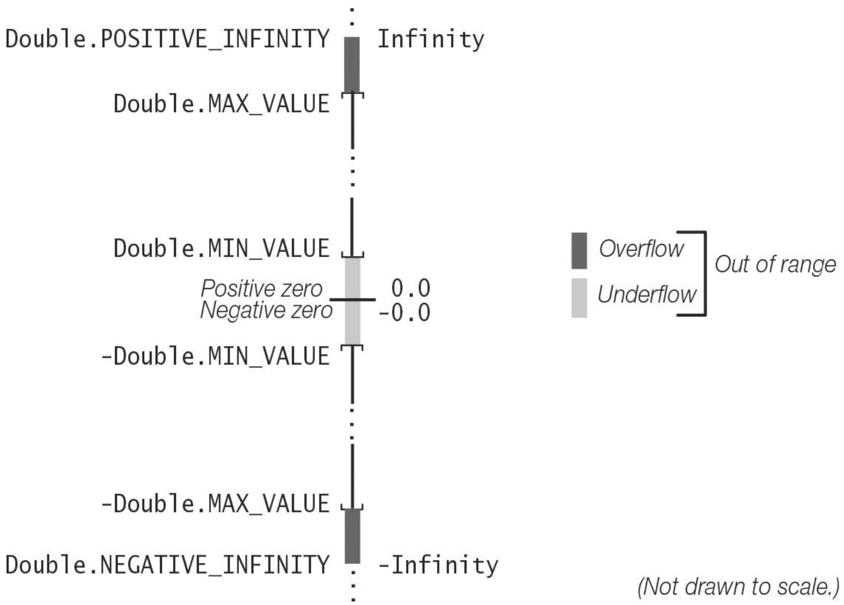
c. Only Java programs can be executed by a JVM.

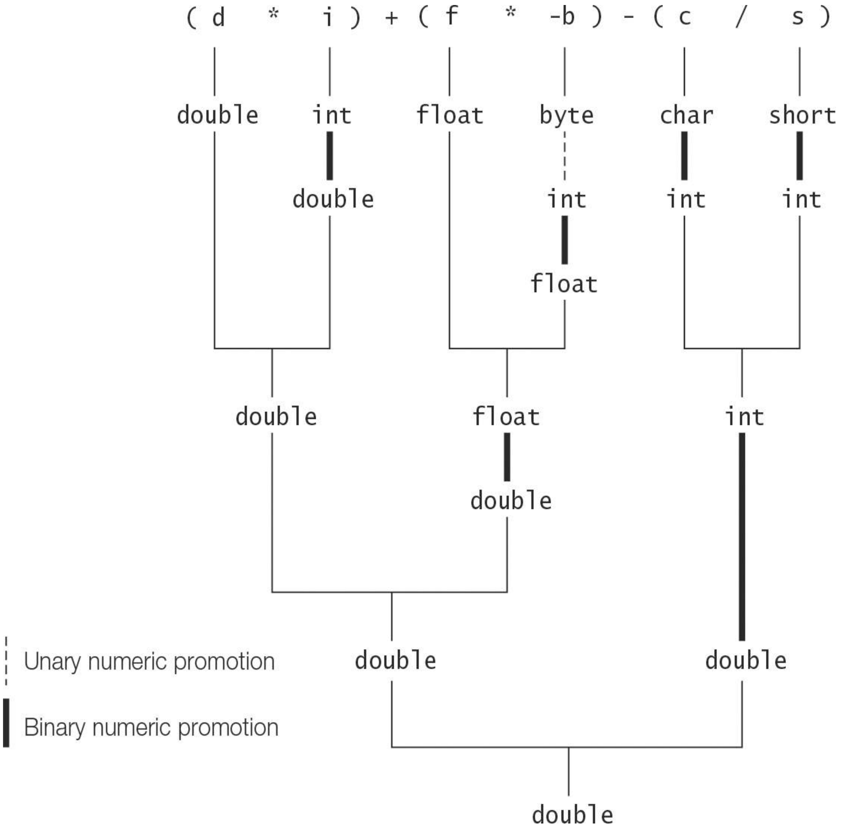
d. A Java program can create and destroy objects.

e. None of the above


Primitive data types
Boolean type
Numeric types
Integral types
Floating-point types
Character type
Integer types
boolean
char
byte
short
int
long
float
double


byte
short
int
long
float
double
char


Double.POSITIVE_INFINITY
Infinity
Double.MAX_VALUE
Double.MIN_VALUE
Positive zero
0.0
Negative zero
-0.0
-Double.MIN_VALUE
-Double.MAX_VALUE
Double.NEGATIVE_INFINITY
-Infinity
Overflow
Underflow
Out of range
(Not drawn to scale.)


(d * i) + (f * -b) - (c / s)
double
int
float
byte
char
short
double
int
int
int
float
double
float
int
double
double
double
double
Unary numeric promotion
Binary numeric promotion

Basic Elements, Primitive Data Types, and Operators 2

Chapter Topics

- Overview of basic language elements in Java: identifiers, keywords, separators, literals, whitespace, and comments
- Overview of primitive data types defined in Java: integral, floating-point, and boolgean
- Representing integers in different number systems and in memory
- Understanding type conversion categories and conversion contexts, and which conversions are permissible in each conversion context
- Defining and evaluating arithmetic and boolean expressions, and the order in which operands and operators are evaluated
- Using Java operators, including precedence and associativity rules for expression evaluation

Java SE 17 Developer Exam O	
[1.1] Use primitives and wrapper classes including Math API, parentheses, type promotion, and casting to evaluate arithmetic and boolean expressions	§2.1, p. 41 to §2.19, p. 92

- *Primitive types, operators, expression evaluation, and type conversions are covered in this chapter.*
- *For wrapper classes, see §8.3, p. 429.*
- *For Math API, see §8.6, p. 478.*

Java SE 11 Developer Exam O	
[1.1] Use primitives and wrapper classes, including, operators, the use of parentheses, type promotion and casting	§2.1, p. 41 to §2.19, p. 92

- *Primitive types, operators, expression evaluation, and type conversions are covered in this chapter.*
- *For wrapper classes, see §8.3, p. 429.*

This chapter covers the low-level language elements from which high-level constructs are formed, the primitive data types that are provided by the language, and the operators that can be used to compose expressions. In addition to how expressions are evaluated, an understanding of which type conversions can be applied in which context is also essential.

2.1 Basic Language Elements

Like any other programming language, the Java programming language is defined by *grammar rules* that specify how *syntactically* legal constructs can be formed using the language elements, and by a *semantic definition* that specifies the *meaning* of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called *lexical tokens* (or just *tokens*) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

Identifiers

A name in a program is called an *identifier*. Identifiers can be used to denote classes, methods, variables, and labels.

In Java, an *identifier* is composed of a sequence of characters, where each character can be either a *letter* or a *digit*. However, the first character in an identifier must always be a letter, as explained later.

Since Java programs are written in the Unicode character set (p. 37), characters allowed in identifier names are interpreted according to this character set. Use of the Unicode character set opens up the possibility of writing identifier names in many writing scripts used around the world. As one would expect, the characters `A` to `Z` and `a` to `z` are letters and the characters `0` to `9` are digits. A *connecting punctuation character* (such as *underscore* _) and any *currency symbol* (such as `$`, `¢`, `¥`, or `£`) are also allowed as letters in identifier names, but these characters should be used judiciously. Note also that the underscore (_) on its own is *not* a legal identifier name, but a keyword (Table 2.1, p. 31).

Table 2.1 *Keywords in Java*

`abstract`	`default`	`if`	`private`	`this`
`assert`	`do`	`implements`	`protected`	`throw`
`boolean`	`double`	`import`	`public`	`throws`
`break`	`else`	`instanceof`	`return`	`transient`
`byte`	`enum`	`int`	`short`	`try`
`case`	`extends`	`interface`	`static`	`void`
`catch`	`final`	`long`	`strictfp`	`volatile`
`char`	`finally`	`native`	`super`	`while`
`class`	`float`	`new`	`switch`	`_` *(underscore)*
`continue`	`for`	`package`	`synchronized`	

Identifiers in Java are *case sensitive*. For example, `price` and `Price` are two different identifiers.

Examples of Legal Identifiers

Click here to view code image

```
number, Number, sum_$, bingo, $$_100, _007, mål, grüß
```

Examples of Illegal Identifiers

Click here to view code image

```
48chevy, all@hands, grand-sum, _
```

The name `48chevy` is not a legal identifier because it starts with a digit. The character `@` is not a legal character in an identifier. It is also not a legal operator, so `all@hands` cannot be interpreted as a legal expression with two operands. The character `-` is not a legal character in an identifier, but it is a legal operator; thus `grand-sum` could be interpreted as a legal expression with two operands. An underscore (_) by itself is not a legal identifier.

Keywords

Keywords are reserved words or identifiers that are predefined in the language and cannot be used to denote other entities. All Java keywords are lowercase, and incorrect usage results in compile-time errors.

Keywords currently defined in the language are listed in Table 2.1. The keyword `strictfp` is obsolete as of Java SE 17, and its use is discouraged in new code. *Contextual keywords* that are restricted in certain contexts are listed in Table 2.2. Keywords currently reserved, but *not in use*, are listed in Table 2.3. In addition, three identifiers are reserved as predefined *literals* in the language: the `null` literal, and the boolean literals `true` and `false` (Table 2.4). A keyword cannot be used as an identifier. A contextual keyword cannot be used as an identifier in certain contexts. The index at the end of the book contains references to relevant sections where currently used keywords are explained.

Table 2.2 *Contextual Keywords*

`exports`	`opens`	`requires`	`uses`
`module`	`permits`	`sealed`	`var`
`non-sealed`	`provides`	`to`	`with`

[illegible]dtive

Table 2.3 *Reserved Keywords Not Currently in Use*

[illegible]

Table 2.4 *Reserved Literals in Java*

[illegible]

Separators

Separators (also known as *punctuators*) are tokens that have meaning depending on the context in which they are used; they aid the compiler in performing syntax and semantic analysis of a program (Table 2.5). The semicolon (`;`) is used to terminate a statement. A pair of curly brackets, `{}`, can be used to group several statements. See the index entries for these separators for more details.

Table 2.5 *Separators in Java*

[illegible]
[illegible] : .

Literals

A *literal* denotes a constant value; in other words, the value that a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean, and string values. In addition, the literal `null` represents the null reference. Table 2.6 shows examples of literals in Java.

Table 2.6 *Examples of Literals*

Integer	[illegible]	
Floating-point	[illegible]	
Character	[illegible]	`')'`
Boolean	[illegible]	
String	[illegible]ce of the action"	

Integer Literals

Integer data types comprise the following primitive data types: `int`, `long`, `byte`, and `short` (p. 41).

The default data type of an integer literal is always `int`, but it can be specified as `long` by appending the suffix `L` (or `l`) to the integer value. The suffix `L` is often preferred because the suffix `l` and the digit `1` can be hard to distinguish. Without the suffix, the `long` literals `2020L` and `0L` will be interpreted as `int` literals. There is no direct way to specify a `short` or a `byte` literal.

In addition to the decimal number system, integer literals can be specified in the binary (*base* 2, *digits* `0`–`1`), octal (*base* 8, *digits* `0`–`7`), and hexadecimal (*base* 16, *digits* `0`–`9` and `a`–`f`) number systems. The digits `a` to `f` in the hexadecimal system correspond to decimal values `10` to `15`. Binary, octal, and hexadecimal numbers are specified with `0b` (or `0B`), `0`, and `0x` (or `0X`) as the base or radix prefix, respectively.

Examples of decimal, binary, octal, and hexadecimal literals are shown in Table 2.7. Note that the leading `0` (zero) digit is not the uppercase letter `O`. The hexadecimal digits from `a` to `f` can also be specified with the corresponding uppercase forms (`A`–`F`). Negative integers (e.g., `-90`) can be specified by prefixing the minus sign (`-`) to the magnitude of the integer regardless of the number system (e.g., `-0b1011010`, `-0132`, or `-0X5a`).

Table 2.7 *Examples of Decimal, Binary, Octal, and Hexadecimal Literals*

Decimal	Binary	Octal	Hexadecimal
`8`	`0b1000`	`010`	`0x8`
`10L`	`0b1010L`	`012L`	`0xaL`
`16`	`0b10000`	`020`	`0x10`
`27`	`0b11011`	`033`	`0x1B`
`90L`	`0b1011010L`	`0132L`	`0x5aL`

`-90`	`-0b1011010`
	or
	`0b11111111111111111111111110100110`
		`-0132`
		or
		`037777777646`
			`-0x5a`
			or
			`0xffffffa6`

`-1`	`-0b1`
	or
	`0b11111111111111111111111111111111`
		`-01`
		or
		`037777777777`
			`-0x1`
			or
			`0xffffffff`

`2147483647` (i.e., $2^{31} - 1$)	`0b01111111111111111111111111111111`	`017777777777`	`0x7fffffff`
`-2147483648` (i.e., -2^{31})	`0b10000000000000000000000000000000`	`020000000000`	`0x80000000`
`1125899906842624L` (i.e., 2^{50})	`0b100000000000000000000000000000000000000000000000000L`	`040000000000000000L`	`0x4000000000000L`

Representing Integers

Integer data types in Java represent *signed* integer values, meaning both positive and negative integer values. The values of type `char` can effectively be regarded as *unsigned* 16-bit integers.

Values of type `byte` are represented as shown in Table 2.8. A value of type `byte` requires 8 bits. With 8 bits, we can represent 2^8 or 256 values. Java uses two's complement (explained later) to store signed values of integer data types. For the `byte` data type, this means values are in the range –128 (i.e., -2^7) to +127 (i.e., $2^7 - 1$), inclusive.

Table 2.8 *Representing Signed* `byte` *Values Using Two's Complement*

Decimal value	**Binary representation (8 bit)**	**Octal value with prefix 0**	**Hexadecimal value with prefix `0x`**
127	`01111111`	`0177`	`0x7f`
126	`01111110`	`0176`	`0x7e`
⋮	⋮	⋮	⋮
41	`00101001`	`051`	`0x29`
⋮	⋮	⋮	⋮
2	`00000010`	`02`	`0x02`
1	`00000001`	`01`	`0x01`
0	`00000000`	`00`	`0x00`
–1	`11111111`	`0377`	`0xff`
–2	`11111110`	`0376`	`0xfe`
⋮	⋮	⋮	⋮
–41	`11010111`	`0327`	`0xd7`
⋮	⋮	⋮	⋮
–127	`10000001`	`0201`	`0x81`
–128	`10000000`	`0200`	`0x80`

Bits in an integral value are usually numbered from right to left, starting with the least significant bit `0` (also called the *rightmost bit*). The representation of the signed types sets the most significant bit to `1`, indicating negative values. Adding `1` to the maximum `int` value `2147483647` results in the minimum value `-2147483648`, such that the values wrap around for integers and no overflow or underflow is indicated.

Calculating Two's Complement

Before we look at two's complement, we need to understand one's complement. The one's complement of a binary integer is computed by inverting the bits in the number. Thus the one's complement of the binary number 00101001 is 11010110. The one's complement of a binary number N_2 is denoted as $\sim N_2$. The following relations hold between a binary integer N_2, its one's complement $\sim N_2$, and its two's complement $-N_2$:

$-N2 = \sim N2 + 10 = -N2 + N2$

If N_2 is a positive binary integer, then $–N_2$ denotes its negative binary value, and vice versa. The second relation states that adding a binary integer N_2 to its two's complement $–N_2$ equals 0.

Given a positive `byte` value, say `41`, the binary representation of `-41` can be found as follows:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Form one's complement, $\sim N_2$:	11010110	
Add 1:	00000001	
Result is two's complement, $–N_2$:	11010111	–41

Adding a number N_2 to its two's complement $–N_2$ gives 0, and the carry bit from the addition of the most significant bits (after any necessary extension of the operands) is ignored:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Add two's complement, $–N_2$:	11010111	–41
Sum:	00000000	0

Subtraction between two integers is also computed as addition with two's complement:

$N_2 – M_2 = N_2 + (–M_2)$

For example, the expression $41_{10} – 3_{10}$ (with the correct result 38_{10}) is computed as follows:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Add $–M_2$ (i.e., subtract M_2):	11111101	–3
Result:	00100110	38

The previous discussion of `byte` values applies equally to values of other integer types: `short`, `int`, and `long`. These types have their values represented by two's complement in 16, 32, and 64 bits, respectively.

Floating-Point Literals

Floating-point data types come in two flavors: `float` and `double`.

The default data type of a floating-point literal is `double`, but it can be explicitly designated by appending the suffix `D` (or `d`) to the value. A floating-point literal can also be specified to be a `float` by appending the suffix `F` (or `f`).

Floating-point literals can also be specified in scientific notation, where `E` (or `e`) stands for *exponent*. For example, the `double` literal `194.9E-2` in scientific notation is interpreted as `194.9` × `10`$^{-2}$ (i.e., `1.949`).

Examples of `double` *Literals*

Click here to view code image

```
0.0         0.0d        0D
0.49        .49         .49D
49.0        49.         49D
4.9E+1      4.9E+1D     4.9e1d     4900e-2    .49E2
```

Examples of float *Literals*

Click here to view code image

```
0.0F        0f
0.49F       .49F
49.0F       49.F        49F
4.9E+1F     4900e-2f    .49E2F
```

Note that the decimal point and the exponent are optional, and that at least one digit must be specified. Also, for the examples of `float` literals presented here, the suffix `F` is mandatory; if it were omitted, they would be interpreted as `double` literals.

Underscores in Numerical Literals

The underscore character (_) can be used to improve the readability of numerical literals in the source code. Any number of underscores can be inserted *between the digits* that make up the numerical literal. This rules out underscores adjacent to the sign (`+`, `-`), the radix prefix (`0b`, `0B`, `0x`, `0X`), the decimal point (`.`), the exponent (`e`, `E`), and the data type suffix (`l`, `L`, `d`, `D`, `f`, `F`), as well as before the first digit and after the last digit. Note that octal radix prefix `0` is part of the definition of an octal literal and is therefore considered the first digit of an octal literal.

Underscores in identifiers are treated as letters. For example, the names `_XXL` and `_XXL_` are two distinct legal identifiers. In contrast, underscores are used as a notational convenience for numerical literals and are ignored by the compiler when used in such literals. In other words, a numerical literal can be specified in the source code using underscores between digits, such that `2_0_2_2` and `20__22` represent the same numerical literal `2022` in source code.

Examples of Legal Use of Underscores in Numerical Literals

Click here to view code image

```
0b0111_1111_1111_1111_1111_1111_1111_1111
0_377_777_777            0xff_ff_ff_ff
-123_456.00              1_2.345_678e1_2
2009__08__13             49_03_01d
```

Examples of Illegal Use of Underscores in Numerical Literals

Click here to view code image

```
_0_b_01111111111111111111111111111111_
_0377777777_             _0_x_ffffffff_
+_123456_._00_           _12_._345678_e_12_
_20090813_               _490301_d_
```

Boolean Literals

The primitive data type `boolean` represents the truth values *true* and *false* that are denoted by the reserved literals `true` and `false`, respectively.

Character Literals

A character literal is quoted in single quotes (`'`). All character literals have the primitive data type `char`.

A character literal is represented according to the 16-bit Unicode character set, which subsumes the 8-bit ISO Latin-1 and the 7-bit ASCII characters. In Table 2.9, note that digits (`0`–`9`), uppercase letters (`A`–`Z`), and lowercase letters (`a`–`z`) have contiguous Unicode values. A Unicode character can always be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix `\u`.

Table 2.9 *Examples of Character Literals*

Character literal using Unicode value	**Character literal**	**Character**
`'\u0020'`	`' '`	*Space*
`'\u0030'`	`'0'`	`0`
`'\u0031'`	`'1'`	`1`
`'\u0039'`	`'9'`	`9`
`'\u0041'`	`'A'`	`A`

`'\u0042'`	`'B'`
`'\u005a'`	`'Z'`
`'\u0061'`	`'a'`
`'\u0062'`	`'b'`
`'\u007a'`	`'z'`
`'\u0084'`	`'Ñ'`
`'\u008c'`	`'å'`
`'\u00a7'`	`'ß'`

Escape Sequences

Certain *escape sequences* define special characters, as shown in Table 2.10. These escape sequences allow representation of some special characters in character literals, string literals (p. 39), and text blocks (§8.4, p. 458). These escape sequences can be single-quoted to define character literals, or included in string literals and text blocks. For example, the escape sequence `\t` and the Unicode value `\u0009` are equivalent. However, the Unicode values `\u000a` and `\u000d` should not be used to represent a newline and a carriage return in the source code. These values are interpreted as line-terminator characters by the compiler and will cause compile-time errors. You should use the escape sequences `\n` and `\r`, respectively, for correct interpretation of these characters in the source code.

Table 2.10 *Escape Sequences*

Escape sequence	Unicode value	Character
`\b`	`\u0008`	Backspace (BS)
`\t`	`\u0009`	Horizontal tab (HT or TAB)
`\n`	`\u000a`	Linefeed (LF), also known as newline (NL)
`\f`	`\u000c`	Form feed (FF)
`\r`	`\u000d`	Carriage return (CR)
`\s`	`\u0020`	Space (SP)
`\`*line terminator*	—	Line continuation in a text block
`\'`	`\u0027`	Apostrophe-quote, also known as single quote
`\"`	`\u0022`	Quotation mark, also known as double quote
`\\`	`\u005c`	Backslash

We can also use the escape sequence `\ddd` to specify a character literal as an octal value, where each digit `d` can be any octal digit (`0`–`7`), as shown in Table 2.11. The number of digits must be three or fewer, and the octal value cannot exceed `\377`; in other words, only the first 256 characters can be specified with this notation.

Table 2.11 *Examples of Escape Sequence* `\ddd`

Escape sequence `\ddd`	Character literal

`'\141'`	`'a'`
`'\46'`	`'&'`
`'\60'`	`'0'`

String Literals

A *string literal* is a sequence of characters that must be enclosed in double quotes and must occur on a single line. All string literals are objects of the class `String` (§8.4, p. 439).

Escape sequences as well as Unicode values can appear in string literals:

Click here to view code image

```
"Here comes a tab.\t And here comes another one\u0009!"          (1)
"What's on the menu?"                                            (2)
"\"String literals are double-quoted.\""                         (3)
"Left!\nRight!"                                                  (4)
"Don't split                                                     (5)
me up!"
```

In (1), the tab character is specified using the escape sequence and the Unicode value, respectively. In (2), the single quote need not be escaped in strings, but it would be if specified as a character literal (`'\''`). In (3), the double quotes in the string must be escaped. In (4), we use the escape sequence `\n` to insert a newline. The expression in (5) generates a compile-time error, as the string literal is split over several lines. Printing the strings from (1) to (4) will give the following result:

Click here to view code image

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
Left!
Right!
```

One should also use the escape sequences `\n` and `\r`, respectively, for correct interpretation of the characters `\u000a` (newline) and `\u000d` (form feed) in string literals.

Whitespace

Whitespace is a sequence of spaces, tabs, form feeds, and line terminator characters in a Java source file. Line terminators include the newline, carriage return, and carriage return–newline sequence.

A Java program is a free-format sequence of characters that is *tokenized* by the compiler—that is, broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes whitespace has to be inserted explicitly as a separator. For example, the identifier `classRoom` will be interpreted as a single token, unless whitespace is inserted to distinguish the keyword `class` from the identifier `Room`.

Whitespace aids not only in separating tokens, but also in formatting the program so that it is easy to read. The compiler ignores the whitespace once the tokens are identified.

Comments

A program can be documented by inserting comments at relevant places in the source code. These comments are for documentation purposes only and are ignored by the compiler.

Java provides three types of comments that can be used to document a program:

- A single-line comment: `// ... to the end of the line`
- A multiple-line comment: `/* ... */`
- A documentation (Javadoc) comment: `/** ... */'`

Single-Line Comment

All characters after the comment-start sequence `//` through to the end of the line constitute a *single-line comment*.

Click here to view code image

```
// This comment ends at the end of this line.
int age;        // From comment-start sequence to the end of the line is a com
```

Multiple-Line Comment

A *multiple-line comment*, as the name suggests, can span several lines. Such a comment starts with the sequence `/*` and ends with the sequence `*/`.

```
/* A comment
   on several
   lines.
*/
```

The comment-start sequences (`//`, `/*`, `/**`) are not treated differently from other characters when occurring within comments, so they are ignored. This means that trying to nest multiple-line comments will result in a compile-time error:

Click here to view code image

```
/* Formula for alchemy.
   gold = wizard.makeGold(stone);
   /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence `/*` is ignored. The last occurrence of the sequence `*/` in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A *documentation comment* is a special-purpose multiple-line comment that is used by the `javadoc` tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods, and field definitions. Special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with the sequence `/**` and ends with the sequence `*/`:

Click here to view code image

```
/**
 *  This class implements a gizmo.
 *  @author K.A.M.
 *  @version 4.0
 */
```

For details on the `javadoc` tool, see the tools documentation provided by the JDK.

2.2 Primitive Data Types

Figure 2.1 gives an overview of the *primitive data types* in Java.

Figure 2.1 *Primitive Data Types in Java*

Primitive data types in Java can be divided into three main categories:

- *Integral types*: represent signed integers (`byte`, `short`, `int`, `long`) and unsigned character values (`char`)
- *Floating-point types* (`float`, `double`): represent fractional

signed numbers

- *Boolean type* (`boolean`): represents logical values

Each primitive data type defines the range of values in the data type, and operations on these values are defined by special operators in the language (p. 51).

Primitive data values are not objects, but each primitive data type has a corresponding *wrapper* class that can be used to represent a primitive value as an object. Wrapper classes are discussed in §8.3, p. 429.

The Integer Types

The integer data types are `byte`, `short`, `int`, and `long` (Table 2.12). Their values are signed integers represented by two's complement (p. 34).

Table 2.12 *Range of Integer Values*

Data type	Width (bits)	Minimum value `MIN_VALUE`	Maximum value `MAX_VALUE`
`byte`	8	-2^7 (`-128`)	2^7-1 (`+127`)
`short`	16	-2^{15} (`-32768`)	$2^{15}-1$ (`+32767`)
`int`	32	-2^{31} (`-2147483648`)	$2^{31}-1$ (`+2147483647`)
`long`	64	-2^{63} (`-9223372036854775808L`)	$2^{63}-1$ (`+9223372036854775807L`)

The `char` Type

The data type `char` represents characters (Table 2.13). Their values are unsigned integers that denote all of the 65,536 (2^{16}) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters.

Table 2.13 *Range of Character Values*

Data type	Width (bits)	Minimum Unicode value	Maximum Unicode value
`char`	16	`0x0 (\u0000)`	`0xffff (\uffff)`

The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

The integer types and the `char` type are collectively called *integral types*.

The Floating-Point Types

Floating-point numbers are represented by the `float` and `double` data types.

Floating-point numbers conform to the IEEE 754-1985 binary floating-point standard. Table 2.14 shows the range of values for positive floating-point numbers, but these apply equally to negative floating-point numbers with the minus sign (-) as a prefix. Zero can be either `0.0` or `-0.0`. The range of values represented by the `double` data type is wider than that of the `float` data type.

Table 2.14 *Range of Floating-Point Values*

Data type	**Width (bits)**	**Minimum positive value `MIN_VALUE`**	**Maximum positive value `MAX_VALUE`**
`float`	32	`1.401298464324817E-45f`	`3.402823476638528860e+38f`
`double`	64	`4.94065645841246544e-324`	`1.79769313486231570e+308`

Since the size for representation is a finite number of bits, certain floating-point numbers can be represented only as approximations. For example, the value of the expression `(1.0/3.0)` is represented as an approximation due to the finite number of bits used to represent floating-point numbers.

The `boolean` Type

The data type `boolean` represents the two logical values denoted by the literals `true` and `false` (Table 2.15).

Table 2.15 *Boolean Values*

Data type	**Width**	**True value literal**	**False value literal**
`boolean`	Not applicable	`true`	`false`

Boolean values are results of all *relational* (p. 74), *conditional* (p. 80), and *boolean* (p. 78) *logical operators*.

Table 2.16 summarizes the pertinent facts about the primitive data types: their width or size, which indicates the number of bits required to store a primitive value; their range of legal values, which is specified by the minimum and maximum values permissible; and the name of the corresponding wrapper class (§8.3, p. 429).

Table 2.16 *Summary of Primitive Data Types*

Data type	**Width (bits)**	**Minimum value, maximum value**	**Wrapper class**
`boolean`	Not applicable	`true, false`	`Boolean`
`byte`	8	-2^7, $2^7 - 1$	`Byte`
`short`	16	-2^{15}, $2^{15} - 1$	`Short`
`char`	16	`0x0, 0xffff`	`Character`

`integer`	$-2^{31}, 2^{31} - 1$	32
`long`	$-2^{63}, 2^{63} - 1$	64
`float`	`±1.40129846432481707e-45f,` `±3.402823476638528860e+38f`	32
`double`	`±4.94065645841246544e-324,` `±1.79769313486231570e+308`	64

2.3 Conversions

In this section we discuss the different kinds of *type conversions* that can be applied to values; in the next section we discuss the *contexts* in which these conversions are permitted. Some type conversions must be *explicitly* stated in the program, while others are performed *implicitly*. Some type conversions can be checked at compile time to guarantee their validity at runtime, while others will require an extra check at runtime.

Widening and Narrowing Primitive Conversions

For the primitive data types, the value of a *narrower* data type can be converted to a value of a *wider* data type. This is called a *widening primitive conversion*. Widening conversions from one primitive type to the next wider primitive type are summarized in Figure 2.2. The conversions shown are transitive. For example, an `int` can be directly converted to a `double` without first having to convert it to a `long` and a `float`.

Figure 2.2 *Widening Primitive Conversions*

Note that the target type of a widening primitive conversion has a *wider range* of values than the source type—for example, the range of the `long` type subsumes the range of the `int` type. In widening conversions between *integral* types, the source value remains intact, with no loss of magnitude information. However, a widening conversion from an `int` or a `long` value to a `float` value, or from a `long` value to a `double` value, may result in a *loss of precision*. The floating-point value in the target type is then a correctly rounded approximation of the integer value. Note that precision relates to the number of significant bits in the value, and must not be confused with *magnitude*, which relates to how large the represented value can be.

Converting from a wider primitive type to a narrower primitive type is called a *narrowing primitive conversion*; it can result in a loss of magnitude information, and possibly in a loss of precision as well. Any conversion that is not a widening primitive conversion according to Figure 2.2 is a narrowing primitive conversion. The target type of a narrowing primitive conversion has a *narrower range* of values than the source type—for example, the range of the `int` type does not include all the values in the range of the `long` type.

Note that all conversions between `char` and the two integer types `byte` and `short` are considered narrowing primitive conversions. The reason is that the conversions between the unsigned type `char` and the signed types `byte` and `short` can result in a loss of information. These narrowing conversions are done in two steps: first converting the source value to the `int` type, and then converting the `int` value to the target type.

Widening primitive conversions are usually done implicitly, whereas narrowing primitive conversions usually require a *cast* (p. 48). It is not illegal to use a cast for a widening conversion. However, the compiler will flag any conversion that requires a cast if none has been specified. Regardless of any loss of magnitude or precision, widening and narrowing primitive conversions *never* result in a runtime exception.

Click here to view code image

```
long year = 2020;    // (1) Implicit widening: long <----- int, assigned 2020L
int pi = (int) 3.14; // (2) Narrowing requires cast: int <----- double, assign
```

Ample examples of widening and narrowing primitive conversions can be found in this chapter.

Widening and Narrowing Reference Conversions

The *subtype–supertype* relationship between reference types determines which conversions are permissible between them (§5.1, p. 191). Conversions *up* the *type hierarchy* are called *widening reference conversions* (also called *upcasting*). Such a conversion converts from a subtype to a supertype:

Click here to view code image

```
Object obj = "Upcast me";  // (1) Widening: Object <----- String
```

Conversions *down* the type hierarchy represent *narrowing reference conversions* (also called *downcasting*):

Click here to view code image

```
String str = (String) obj; // (2) Narrowing requires cast: String <----- Objec
```

A subtype is a *narrower* type than its supertype in the sense that it is a specialization of its supertype. Contexts under which reference conversions can occur are discussed in §5.8, p. 261.

Widening reference conversions are usually done implicitly, whereas narrowing reference conversions usually require a cast, as illustrated in the second declaration statement above. The compiler will reject casts that are not legal or will issue an *unchecked warning* under certain circumstances if type-safety cannot be guaranteed.

Widening reference conversions do not require any runtime checks and never result in an exception during execution. This is not the case for narrowing reference conversions, which require a runtime check and can throw a `ClassCastException` if the conversion is not legal.

Boxing and Unboxing Conversions

Boxing and unboxing conversions allow interoperability between primitive values and their representation as objects of the wrapper types (§8.3, p. 429).

A *boxing conversion* converts the value of a primitive type to a corresponding value of its wrapper type, and an *unboxing conversion* converts the value of a wrapper type to a value of its corresponding primitive type. Both boxing and unboxing conversion are applied implicitly in the right context, but the wrapper classes also provide the `static` method `valueOf()` to explicitly box a primitive value in a wrapper object, and the method *primitiveType*`Value()` to explicitly unbox the value in a wrapper object as a value of *primitiveType*.

Click here to view code image

```
Integer iRef = 10;                   // (1) Implicit boxing: Integer <----- in
Double dRef = Double.valueOf(3.14);  // (2) Explicit boxing: Double <----- dou

int i = iRef;                      // (3) Implicit unboxing: int <----- Intege
double d = dRef.doubleValue();     // (4) Explicit unboxing: double <----- Dou
```

At (1) above, the `int` value `10` results in an object of type `Integer` implicitly being created; this object contains the `int` value `10`. We say that the `int` value `10` has been *boxed* in an object of the wrapper type `Integer`. This implicit boxing conversion is also called *autoboxing*. An explicit boxing by the `valueOf()` method of the wrapper classes is used at (2) to box a `double` value.

Unboxing conversion is illustrated by (3) and (4) above. Implicit unboxing is applied at (3) to unbox the value in the `Integer` object, and explicit unboxing is applied at (4) by calling the `doubleValue()` method of the `Double` class.

Note that both boxing and unboxing are done implicitly in the right context. Boxing allows primitive values to be used where an object of their wrapper type is expected, and unboxing allows the converse. Unboxing makes it possible to use a `Boolean` wrapper object as a `boolean` value in a boolean expression, and to use an integral wrapper object as an integral primitive value in an arithmetic expression. Unboxing a wrapper reference that has the `null` value results in a `NullPointer-Exception`. Ample examples of boxing and unboxing can be found in this chapter and in §5.8, p. 261.

Other Conversions

Here we briefly mention some other conversions.

- *Identity conversions* allow conversions from a type to that same type. An identity conversion is always permitted.

  Click here to view code image

  ```
  int i = (int) 10;                  // int <---- int
  String str = (String) "Hi";        // String <---- String
  ```

- *String conversions* allow a value of any other type to be converted to a `String` type in the context of the string concatenation operator `+` (p. 67).
- *Unchecked conversions* are permitted to facilitate operability between legacy and generic code (§11.2, p. 575).

2.4 Type Conversion Contexts

Selected conversion contexts and the conversions that are applicable in these contexts are summarized in Table 2.17. The conversions shown in each context occur *implicitly*, without the program having to take any special action. For other conversion contexts, see §2.3, p. 46.

Table 2.17 *Selected Conversion Contexts and Conversion Categories*

	Conversion contexts

Assignment	Method invocation	Casting	Numeric promotion

Widening/narrowing *primitive* conversions

Both

Narrowing for *constant expressions* of non-`long` integral type, with optional boxing

Widening/narrowing *reference* conversions

Not applicable

Both, followed by optional unchecked conversion

Boxing/unboxing

Unboxing, followed by optional widening *primitive* conversion

Both

Boxing, followed by optional widening *reference* conversion

Assignment Context

Assignment conversions that can occur in an assignment context are shown in the second column of Table 2.17. An assignment conversion converts the type of an expression to the type of a target variable.

An expression (or its value) is *assignable* to the target variable, if the type of the expression can be converted to the type of the target variable by an assignment conversion. Equivalently, the type of the expression is *assignment compatible* with the type of the target variable.

For assignment conversion involving primitive data types, see §2.7, p. 54. Note the special case where a narrowing conversion occurs when assigning a non-`long` integer constant expression:

Click here to view code image

```
byte b = 10;   // Narrowing conversion: byte <--- int
```

For assignment conversions involving reference types, see §5.8, p. 261.

Method Invocation Context

Method invocation conversions that can occur in a method invocation context are shown in the third column of Table 2.17. Note that method invocation and assignment conversions differ in one respect: Method invocation conversions do not include the implicit narrowing conversion performed for non-`long` integral constant expressions.

Click here to view code image

```
// Assignment: (1) Implicit narrowing followed by (2) boxing.
Character space1 = 32;     // Character <-(2)-- char <-(1)-- int

// Invocation of method with signature: valueOf(char)
Character space2 = Character.valueOf(32);       // Compile-time error!
```

```
                                                // Call signature: valueOf(int
Character space3 = Character.valueOf((char)32); // OK!
                                                // Call signature: valueOf(cha
```

A method invocation conversion involves converting each argument value in a method or constructor call to the type of the corresponding formal parameter in the method or constructor declaration.

Method invocation conversions involving parameters of primitive data types are discussed in §3.10, p. 129, and those involving reference types are discussed in §5.8, p. 261.

Casting Context of the Unary Type Cast Operator (*type*)

Java, being a *strongly typed* language, checks for *type compatibility* (i.e., it checks whether a type can substitute for another type in a given context) at compile time. However, some checks are possible only at runtime (e.g., which type of object a reference actually denotes during execution). In cases where an operator would have incompatible operands (e.g., assigning a `double` to an `int`), Java demands that a *type cast* be used to *explicitly* indicate the type conversion. The type cast construct has the following syntax:

(*type*) *expression*

The *cast operator* (*type*) is applied to the value of the *expression*. At runtime, a cast results in a new value of *type,* which best represents the value of the *expression* in the old type. We use the term *casting* to mean applying the cast operator for *explicit* type conversion.

However, in the context of casting, *implicit* casting conversions can take place. These casting conversions are shown in the fourth column of Table 2.17. Casting conversions include more conversion categories than the assignment or the method invocation conversions. In the code that follows, the comments indicate the category of the conversion that takes place because of the cast operator on the right-hand side of each assignment—although casts are only necessary for the sake of the assignment at (1) and (2).

Click here to view code image

```
long l = (long) 10;          // Widening primitive conversion: long <--- int
int i = (int) l;             // (1) Narrowing primitive conversion: int <--- l
Object obj = (Object) "7Up"; // Widening ref conversion: Object <--- String
String str = (String) obj;   // (2) Narrowing ref conversion: String <--- Obje
Integer iRef = (Integer) i;  // Boxing: Integer <--- int
```

```
   i = (int) iRef;              // Unboxing: int <--- Integer
```

A casting conversion is applied to the value of the operand *expression* of a cast operator. Casting can be applied to primitive values as well as references. Casting between primitive data types and reference types is not permitted, except where boxing and unboxing is applicable. Boolean values cannot be cast to other data values, and vice versa. The reference literal `null` can be cast to any reference type.

Examples of casting between primitive data types are provided in this chapter. Casting reference values is discussed in §5.11, p. 269.

Numeric Promotion Context

Numeric operators allow only operands of certain types. Numeric promotion results in conversions being applied to the operands to convert them to permissible types. *Numeric promotion conversions* that can occur in a numeric promotion context are shown in the fifth column of Table 2.17. Permissible conversion categories are widening primitive conversions and unboxing conversions. A distinction is made between unary and binary numeric promotion.

Unary Numeric Promotion

Unary numeric promotion proceeds as follows:

- If the single operand is of type `Byte`, `Short`, `Character`, or `Integer`, it is unboxed. If the resulting value is narrower than `int`, it is promoted to a value of type `int` by a widening conversion.
- Otherwise, if the single operand is of type `Long`, `Float`, or `Double`, it is unboxed.
- Otherwise, if the single operand is of a type narrower than `int`, its value is promoted to a value of type `int` by a widening conversion.
- Otherwise, the operand remains unchanged.

In other words, *unary numeric promotion results in an operand value that is either* `int` *or wider.*

Unary numeric promotion is applied in the following expressions:

- Operand of the unary arithmetic operators `+` and `-` (p. 58)

- Array creation expression; for example, `new int[20]`, where the dimension expression (in this case, `20`) must evaluate to an `int` value (§3.9, p. 117)
- Indexing array elements; for example, `objArray['a']`, where the index expression (in this case, `'a'`) must evaluate to an `int` value (§3.9, p. 120)

Binary Numeric Promotion

Binary numeric promotion implicitly applies appropriate widening primitive conversions so that the widest numeric type of a pair of operands is always at least `int`. If `T` is the widest numeric type of two operands after any unboxing conversions have been performed, the operands are promoted as follows during binary numeric promotion:

> If `T` is wider than `int`, both operands are converted to `T`; otherwise, both operands are converted to `int`.

This means that *the resulting type of the operands is at least* `int`.

Binary numeric promotion is applied in the following expressions:

- Operands of the arithmetic operators `*`, `/`, `%`, `+`, and `-` (p. 58)
- Operands of the relational operators `<`, `<=`, `>`, and `>=` (p. 74)
- Operands of the numerical equality operators `==` and `!=` (p. 75)
- Operands of the conditional operator `? :`, under certain circumstances (p. 90)

2.5 Precedence and Associativity Rules for Operators

Precedence and associativity rules are necessary for deterministic evaluation of expressions. The operators are summarized in Table 2.18. The majority of them are discussed in subsequent sections in this chapter. See also the index entries for these operators.

Table 2.18 *Operator Summary*

`[expression] . (args)`	Array element access, member access, method invocation
`expression ++ expression--`	

Unary postfix operators	
Unary prefix operators	`~ ! ++`*expression* `--`*expression* `+`*expression* `-`*expression*
Unary prefix creation and cast	`new (`*type*`)`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< <= > >= instanceof`
Equality	`== !=`
Bitwise/logical AND	`&`
Bitwise/logical XOR	`^`
Bitwise/logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Arrow operator	`->`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

The following remarks apply to Table 2.18:

- The operators are shown with decreasing precedence from the top of the table.
- Operators within the same row have the same precedence.
- Parentheses, `()`, can be used to override precedence and associativity.
- The *unary operators*, which require one operand, include the following: the postfix increment (`++`) and decrement (`--`) operators from the first row, all the prefix operators (`+`, `-`, `++`, `--`, `~`, `!`) in the second row, and the prefix operators (object creation operator `new`, cast operator `(`*type*`)`) in the third row.
- The conditional operator (`? :`) is *ternary*—that is, it requires three operands.
- All operators not identified previously as unary or ternary are *binary*—that is, they require two operands.
- All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
- Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.

Depending on the context, brackets (`[]`), parentheses (`()`), the colon (`:`), and the dot operator (`.`) can also be interpreted as *separators* (p. 32). See the index entries for these separators for more details.

Precedence rules are used to determine which operator should be applied first if there are two operators with a *different* precedence, and these operators follow each other in the expression. In such a case, the operator with the highest precedence is applied first.

The expression `2 + 3 * 4` is evaluated as `2 + (3 * 4)` (with the result `14`) since `*` has higher precedence than `+`.

Associativity rules are used to determine which operator should be applied first if there are two operators with the *same* precedence, and these operators follow each other in the expression.

Left associativity implies grouping from left to right: The expression `7 - 4 + 2` is interpreted as `((7 - 4) + 2)`, since the binary operators `+` and `-` both have the same precedence and left associativity.

Right associativity implies grouping from right to left: The expression `- - 4` is interpreted as `(- (- 4))` (with the result `4`), since the unary operator `-` has right associativity.

The precedence and associativity rules together determine the *evaluation order of the operators*.

2.6 Evaluation Order of Operands

To understand the result returned by an operator, it is important to understand the *evaluation order of its operands*. In general, the operands of operators are evaluated from left to right. The evaluation order also respects any parentheses, and the precedence and associativity rules of operators.

Examples illustrating how the operand evaluation order influences the result returned by an operator can be found in §2.7, p. 54, and §2.10, p. 69.

Left-Hand Operand Evaluation First

The left-hand operand of a binary operator is fully evaluated before the right-hand operand is evaluated.

The evaluation of the left-hand operand can have side effects that can influence the value of the right-hand operand. For example, in the code

Click here to view code image

```
int b = 10;
System.out.println((b=3) + b);
```

the value printed will be `6` and not `13`. The evaluation proceeds as follows:

Click here to view code image

```
    (b=3) + b
     3   + b          b is assigned the value 3
     3   + 3
     6
```

If evaluation of the left-hand operand of a binary operator throws an exception (§7.1, p. 365), we cannot rely on the presumption that the right-hand operand has been evaluated.

Operand Evaluation before Operation Execution

Java guarantees that *all* operands of an operator are fully evaluated *before* the actual operation is performed. This rule does *not* apply to the short-circuit conditional operators `&&`, `||`, and `?:`.

This rule also applies to operators that throw an exception (the integer division operator `/` and the integer remainder operator `%`). The operation is performed only if the operands evaluate normally. Any side effects of the right-hand operand will have been effectuated before the operator throws an exception.

Example 2.1 illustrates the evaluation order of the operands and precedence rules for arithmetic expressions. We use the `eval()` method at (3) in Example 2.1 to demonstrate integer expression evaluation. The first argument to this method is the operand value that is returned by the method, and the second argument is a string to identify the evaluation order.

The argument to the `println()` method in the statement at (1) is an integer expression to evaluate 2 + 3 * 4. The evaluation of each operand in the expression at (1) results in a call of the `eval()` method declared at (3).

Click here to view code image

```
out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));  // (1)
```

The output from Example 2.1 shows that the operands were evaluated first, from left to right, before operator execution, and that the expression was evaluated as `(2 + (3 * 4))`, respecting the precedence rules for arithmetic expression evaluation. Note how the value of variable `j` changes successively from left to right as the first two operands are evaluated.

Example 2.1 *Evaluation Order of Operands and Arguments*

Click here to view code image

```
import static java.lang.System.out;

public class EvalOrder{
  public static void main(String[] args){

    int j = 2;
    out.println("Evaluation order of operands:");
    out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));    // (1

    int i = 1;
    out.println("Evaluation order of arguments:");
    add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n")); // (2) Three argume
  }

  public static int eval(int operand, String str) {        // (3)
    out.print(operand + str);       // Print int operand and String str.
    return operand;                 // Return int operand.
  }

  public static void add3(int operand1, int operand2, int operand3) {    // (4
    out.print(operand1 + operand2 + operand3);
  }
}
```

Output from the program:

Click here to view code image

```
Evaluation order of operands:
2 + 3 * 4
14
Evaluation order of arguments:
1, 2, 3
6
```

Left-to-Right Evaluation of Argument Lists

In a method or constructor invocation, each argument expression in the argument list is fully evaluated before any argument expression to its right.

If evaluation of an argument expression does not complete normally, we cannot presume that any argument expression to its right has been evaluated.

We can use the `add3()` method at (4) in Example 2.1, which takes three arguments, to demonstrate the order in which the arguments in a method call are evaluated. The method call at (2)

Click here to view code image

```
add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n"));  // (2) Three arguments
```

results in the following output, clearly indicating that the arguments were evaluated from left to right, before being passed to the method:

```
1, 2, 3
6
```

Note how the value of variable `i` changes successively from left to right as the first two arguments are evaluated.

2.7 The Simple Assignment Operator `=`

The assignment statement has the following syntax:

variable = *expression*

which can be read as "the target, *variable*, gets the value of the source, *expression*." The previous value of the target variable is overwritten by the assignment operator `=`.

The target *variable* and the source *expression* must be assignment compatible. The target variable must also have been declared. Since variables can store either primitive values or reference values, *expression* evaluates to either a primitive value or a reference value.

Assigning Primitive Values

The following examples illustrate assignment of primitive values:

Click here to view code image

```
int j, k;
j = 0b10;          // j gets the value 2.
j = 5;             // j gets the value 5. Previous value is overwritten.
k = j;             // k gets the value 5.
```

The assignment operator has the lowest precedence, so the expression on the right-hand side is evaluated before the assignment is done.

Click here to view code image

```
int i;
i = 5;            // i gets the value 5.
i = i + 1;        // i gets the value 6. + has higher precedence than =.
i = 20 - i * 2;   // i gets the value 8: (20 - (i * 2))
```

Assigning References

Copying reference values by assignment creates *aliases*. Below, the variable `pizza1` is a reference to a pizza that is hot and spicy, and `pizza2` is a reference to a pizza that is sweet and sour.

Click here to view code image

```
Pizza pizza1 = new Pizza("Hot&Spicy");
Pizza pizza2 = new Pizza("Sweet&Sour");

pizza2 = pizza1;
```

Assigning `pizza1` to `pizza2` means that `pizza2` now refers to the same pizza as `pizza1`, the hot and spicy one. After the assignment, these variables are aliases and either one can be used to manipulate the hot and spicy `Pizza` object.

Assigning a reference value does *not* create a copy of the source object denoted by the reference variable on the right-hand side. It merely assigns the reference value of the variable on the right-hand side to the variable on the left-hand side so that they denote the same object. Reference assignment also does not copy the *state* of the source object to any object denoted by the reference variable on the left-hand side.

A more detailed discussion of reference assignment can be found in §5.8, p. 261.

Multiple Assignments

The assignment statement is an *expression statement*, which means that application of the binary assignment operator returns the value of the expression on the *right-hand* side.

Click here to view code image

```
int j, k;
j = 10;      // (1) j gets the value 10, which is returned
k = j;       // (2) k gets the value of j, which is 10, and this value is retur
```

The value returned by an assignment statement is usually discarded, as in the two assignment statements above. We can verify the value returned as follows:

Click here to view code image

```
System.out.println(j = 10);   // j gets the value 10, which is printed.
System.out.println(k = j);    // k gets the value of j, i.e. 10, which is prin
```

The two assignments (1) and (2) above can be written as multiple assignments, illustrating the right associativity of the assignment operator:

Click here to view code image

```
k = j = 10;     // (k = (j = 10))
```

Multiple assignments are equally valid with references:

Click here to view code image

```
Pizza pizzaOne, pizzaTwo;
pizzaOne = pizzaTwo = new Pizza("Supreme"); // Aliases
```

The following example shows the effect of operand evaluation order:

Click here to view code image

```
int[] a = {10, 20, 30, 40, 50}; // An array of int (§3.9, p. 119)
int index = 4;
a[index] = index = 2;           // (1)
```

What is the value of `index`, and which array element `a[index]` is assigned a value in the multiple assignment statement at (1)? The evaluation proceeds as follows:

Click here to view code image

```
a[index] = index = 2;
a[4]     = index = 2;
a[4]     = (index = 2);        // index gets the value 2. = is right associati
a[4]     =      2;             // The value of a[4] is changed from 50 to 2.
```

The following declaration statement will not compile, as the variable `v2` has not been declared:

Click here to view code image

```
int v1 = v2 = 2016;            // Only v1 is declared. Compile-time error!
```

Type Conversions in an Assignment Context

If the target and the source have the same type in an assignment, then obviously the source and the target are assignment compatible and the source value need not be converted. Otherwise, if a widening primitive conversion is permissible, then the widening conversion is applied implicitly; that is, the source type is converted to the target type in an assignment context.

Click here to view code image

```
// Widening Primitive Conversions
int    smallOne = 1234;             // No widening necessary.
long   bigOne   = 2020;             // Widening: int to long.
double largeOne = bigOne;           // Widening: long to double.
double hugeOne  = (double) bigOne;  // Cast redundant but allowed.
```

A widening primitive conversion can result in loss of *precision*. In the next example, the precision of the least significant bits of the `long` value may be lost when it is converted to a `float` value:

Click here to view code image

```
long bigInteger = 98765432112345678L;
float fpNum = bigInteger;  // Widening but loss of precision: 9.8765436E16
```

Additionally, *implicit narrowing primitive conversions* on assignment can occur in cases where *all* of the following conditions are fulfilled:

- The source is a *constant expression* of type `byte`, `short`, `char`, or `int`.
- The target type is of type `byte`, `short`, or `char`.
- The value of the source is determined to be in the range of the target type at compile time.

A *constant expression* is an expression that denotes either a primitive or a `String` literal; it is composed of operands that can be only *literals* or *constant variables*, and operators that can be evaluated only at compile time (e.g., arithmetic and numerical comparison operators, but not increment/decrement operators and method calls). A *constant variable* is a `final` variable of either a primitive type or the `String` type that is initialized with a constant expression.

Click here to view code image

```
int result = 100;               // Not a constant variable. Not declared final
final char finalGrade = 'A';    // Constant variable. ’A’

System.out.printf("%d%n%s%n%d%n%.2f%n%b%n%d%n%d%n",
    2022,                       // Constant expression. 2022
    "Trust " + "me!",           // Constant expression. "Trust me"
    2 + 3 * 4,                  // Constant expression. 14
    Math.PI * Math.PI * 10.0,   // Constant expression. 98.70
    finalGrade == 'A',          // Constant expression. true
    Math.min(2020, 2021),       // Not constant expression. Method call.
    ++result                    // Not constant expression. Increment operator
);
```

Here are some examples that illustrate how the conditions mentioned previously affect narrowing primitive conversions:

Click here to view code image

```
// Conditions fulfilled for implicit narrowing primitive conversions.
short s1 = 10;       // int value in range.
short s2 = 'a';      // char value in range.
char c1 = 32;        // int value in range.
char c2 = (byte)35;  // byte value in range. (int value in range, without cast
byte b1 = 40;        // int value in range.
byte b2 = (short)40; // short value in range. (int value in range, without cas
final int i1 = 20;   // Constant variable
byte b3 = i1;        // final value of i1 in range.
```

All other narrowing primitive conversions will produce a compile-time error on assignment and will explicitly require a cast. Here are some examples:

Click here to view code image

```
// Conditions not fulfilled for implicit narrowing primitive conversions.
// A cast is required.
int i2 = -20;            // i2 is not a constant variable. i2 is not final.
final int i3 = i2;       // i3 is not a constant variable, since i2 is not.
final int i4 = 200;      // i4 is a constant variable.
final int i5;            // i5 is not a constant variable.
short s3 = (short) i2;   // Not constant expression.
char  c3 = (char)  i3;   // Final value of i3 not determinable at compile time
char  c4 = (char)  i2;   // Not constant expression.
byte  b4 = (byte)  128;  // int value not in range.
byte  b5 = (byte)  i4;   // Value of constant variable i4 is not in range.
i5 = 100;                // Initialized at runtime.
short s4 = (short) i5;   // Final value of i5 not determinable at compile time
```

Floating-point values are truncated when cast to integral values.

Click here to view code image

```
// The value is truncated to fit the size of the target type.
float huge   = (float) 1.7976931348623157d;  // double to float.
long  giant  = (long)  4415961481999.03D;    // (1) double to long.
int   big    = (int)   giant;                // (2) long to int.
short small  = (short) big;                  // (3) int to short.
byte  tiny   = (byte)  small;                // (4) short to byte.
char  symbol = (char)  112.5F;               // (5) float to char.
```

Table 2.19 shows how the values are truncated for assignments from (1) to (5).

Table 2.19 *Examples of Truncated Values*

Decimal Binary
4(1)5961481999000000000100000000100001010111101000011000011000 01111
7(8)5101071111101000011000011000 01111
-(1)5601 1100001111
1(5)001111
'(p)000000001110000

The discussion of numeric assignment conversions also applies to numeric parameter values at method invocation (§3.10, p. 129), except for the narrowing conversions, which always require a cast.

The following examples illustrate boxing and unboxing in an assignment context:

Click here to view code image

```
Boolean   boolRef = true;   // Boxing.
Byte      bRef =  2;        // Constant in range: narrowing, then boxing.
// Byte  bRef2 =  257;      // Constant not in range. Compile-time error!

short s = 10;               // Narrowing from int to short.
// Integer   iRef1 = s;     // short not assignable to Integer.
Integer iRef3 = (int) s;    // Explicit widening with cast to int and boxing

boolean bv1 = boolRef;      // Unboxing.
byte  b1 = bRef;            // Unboxing.
int   iVal = bRef;          // Unboxing and widening.

Integer iRefVal = null;            // Always allowed.
// int j = iRefVal;                // NullPointerException at runtime.
if (iRef3 != null) iVal = iRef3;   // Avoids exception at runtime.
```

2.8 Arithmetic Operators: `*`, `/`, `%`, `+`, `-`

Arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of a numeric type (which includes the `char` type).

Floating-point operations are now consistently *strict*; that is, they are executed in accordance with the IEEE-754 32-bit (`float`) and 64-bit (`double`) standard formats. This means that floating-point arithmetic operations give the same results on any JVM implementation. The keyword `strictfp` used to enforce strict behavior for floating-point arithmetic is now obsolete and should not be used in new code.

Arithmetic Operator Precedence and Associativity

In Table 2.20, the precedence of the operators appears in decreasing order, starting from the top row, which has the highest precedence. Unary subtraction has higher precedence than multiplication. The operators in the same row have the same precedence. Binary multiplication, division, and remainder operators have the same precedence. The unary operators have right associativity, and the binary operators have left associativity.

Table 2.20 *Arithmetic Operators*

Unary	*Minus*	*Plus*	
Binary	*Multiplication*	*Division*	*Remainder*
	Addition	*Subtraction*	

Evaluation Order in Arithmetic Expressions

Java guarantees that the operands are fully evaluated from left to right before an arithmetic binary operator is applied. If evaluation of an operand results in an error, the subsequent operands will not be evaluated.

In the expression `a + b * c`, the operand `a` will always be fully evaluated before the operand `b`, which will always be fully evaluated before the operand `c`. However, the multiplication operator `*` will be applied before the addition operator `+`, respecting the precedence rules. Note that `a`, `b`, and `c` are arbitrary arithmetic expressions that have been determined to be the operands of the operators.

Example 2.1, p. 53, illustrates the evaluation order and precedence rules for arithmetic expressions.

Range of Numeric Values

As we have seen, all numeric types have a range of valid values (p. 41). This range is given by the constants named `MAX_VALUE` and `MIN_VALUE`, which are defined in each numeric wrapper type.

The arithmetic operators are overloaded, meaning that the operation of an operator varies depending on the type of its operands. Floating-point arithmetic is performed if any operand of an operator is of floating-point type; otherwise, integer arithmetic is performed.

Values that are out of range or are the results of invalid expressions are handled differently depending on whether integer or floating-point arithmetic is performed.

Integer Arithmetic

Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which cause an `ArithmeticException` (see the later discussion of the division operator `/` and the remainder operator `%`). A valid value does not necessarily mean that the result is correct, as demonstrated by the following examples:

Click here to view code image

```
int tooBig   = Integer.MAX_VALUE + 1;     // -2147483648 which is Integer.MIN_V
int tooSmall = Integer.MIN_VALUE - 1;     //  2147483647 which is Integer.MAX_V
```

These results should be values that are out of range. However, integer arithmetic *wraps around* the result if it is out of range; that is, the result is reduced modulo in the range of the result type. To avoid wrapping around out-of-range values, programs should use either explicit checks or a wider type. If the type `long` were used in the earlier examples, the results would be correct in the `long` range:

Click here to view code image

```
long notTooBig   = Integer.MAX_VALUE + 1L;    //  2147483648L in range.
long notTooSmall = Integer.MIN_VALUE - 1L;    // -2147483649L in range.
```

Floating-Point Arithmetic

Certain floating-point operations result in values that are out of range. Typically, adding or multiplying two very large floating-point numbers can result in an out-of-range value that is represented by *infinity* (Figure 2.3). Attempting floating-point division by zero also returns infinity. The following examples show how this value is printed as signed infinity:

Click here to view code image

```
System.out.println( 4.0 / 0.0);          // Prints:  Infinity
System.out.println(-4.0 / 0.0);          // Prints: -Infinity
```

Both positive and negative infinity represent *overflow* to infinity; that is, the value is too large to be represented as a `double` or `float` (Figure 2.3). Signed infinity is represented by the named constants `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` in the wrapper classes `java.lang.Float` and `java.lang.Double`. A value can be compared with these constants to detect overflow.

Figure 2.3 *Overflow and Underflow in Floating-Point Arithmetic*

Floating-point arithmetic can also result in *underflow* to zero, when the value is too small to be represented as a `double` or `float` (Figure 2.3). Underflow occurs in the following situations:

- The result is between `Double.MIN_VALUE` (or `Float.MIN_VALUE`) and zero, as with the result of `(5.1E-324 - 4.9E-324)`. Underflow then returns positive zero `0.0` (or `0.0F`).
- The result is between `-Double.MIN_VALUE` (or `-Float.MIN_VALUE`) and zero, as with the result of `(-Double.MIN_VALUE * 1E-1)`. Underflow then returns negative zero `-0.0` (or `-0.0F`).

Negative zero compares equal to positive zero; in other words, `(-0.0 == 0.0)` is `true`.

Certain operations have no mathematical result, and are represented by *NaN* (*Not-a-Number*). For example, calculating the square root of –1 results in a NaN. Another example is (floating-point) dividing zero by zero:

Click here to view code image

```
System.out.println(0.0 / 0.0); // Prints: NaN
```

NaN is represented by the constant named `NaN` in the wrapper classes `java.lang.Float` and `java.lang.Double`. Any operation involving NaN produces NaN. Any comparison (except inequality, `!=`) involving NaN and any other value (including NaN) returns `false`. An inequality comparison of NaN with another value (including NaN) always returns `true`. However, the recommended way of checking a value for NaN is to use the static method `isNaN()` defined in both wrapper classes, `java.lang.Float` and `java.lang.Double`.

Unary Arithmetic Operators: `-`, `+`

The unary operators have the highest precedence of all the arithmetic operators. The unary operator `-` negates the numeric value of its operand. The following example illustrates the right associativity of the unary operators:

Click here to view code image

```
int value = - -10; // (-(-10)) is 10
```

Notice the blank space needed to separate the unary operators; without the blank space, these would be interpreted as the decrement operator `--` (p. 69), which would result in a compile-time error because a literal cannot be decremented. The unary operator + has no effect on the evaluation of the operand value.

Multiplicative Binary Operators: `*`, `/`, `%`

Multiplication Operator: `*`

The multiplication operator * multiplies two numbers, as one would expect.

Click here to view code image

```
int    sameSigns     = -4    * -8;   // result:  32
double oppositeSigns =  4    * -8.0; // Widening of int 4 to double. result: -
int    zero          =  0    * -0;   // result:   0
```

Division Operator: `/`

The division operator `/` is overloaded. If its operands are integral, the operation results in *integer division*.

Click here to view code image

```
int    i1 = 4  / 5;    // result: 0
int    i2 = 8  / 8;    // result: 1
double d1 = 12 / 8;    // result: 1.0; integer division, then widening conversi
```

Integer division always returns the quotient as an integer value; that is, the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type. The integer value is subjected to a widening conversion in the assignment context.

An `ArithmeticException` is thrown when integer division with zero is attempted, meaning that integer division by zero is an illegal operation.

If any of the operands is a floating-point type, the operation performs *floating-point division*, where relevant operand values undergo binary numeric promotion:

Click here to view code image

```
double d2 = 4.0 / 8;      // result: 0.5
double d3 = 8 / 8.0;      // result: 1.0
```

```
float d4  = 12.0F / 8;    // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;       // ((12.0 / 4.0) * 3.0) which is 9.0
double result2 = 12.0 * 3.0 / 4.0;       // ((12.0 * 3.0) / 4.0) which is 9.0
```

Remainder Operator: `%`

In mathematics, when we divide a number (the *dividend*) by another number (the *divisor*), the result can be expressed in terms of a *quotient* and a *remainder*. For example, when 7 is divided by 5, the quotient is 1 and the remainder is 2. The remainder operator `%` returns the remainder of the division performed on the operands.

Click here to view code image

```
int quotient  = 7 / 5;   // Integer division operation: 1
int remainder = 7 % 5;   // Integer remainder operation: 2
```

For *integer remainder operation*, where only integer operands are involved, evaluation of the expression `(x % y)` always satisfies the following relation:

$$x == (x / y) * y + (x \% y)$$

In other words, the right-hand side yields a value that is always equal to the value of the dividend. The following examples show how we can calculate the remainder so that this relation is satisfied:

Calculating (7 % 5):

Click here to view code image

```
7 == (7 / 5) * 5 + (7 % 5)
  == (  1  ) * 5 + (7 % 5)
  ==           5 + (7 % 5)
2 ==               (7 % 5)           (7 % 5) is equal to 2
```

Calculating (7 % –5):

Click here to view code image

```
7 == (7 / -5) * -5 + (7 % -5)
  == (  -1  ) * -5 + (7 % -5)
  ==             5 + (7 % -5)
2 ==                 (7 % -5)        (7 % -5) is equal to 2
```

Calculating (–7 % 5):

Click here to view code image

```
-7 == (-7 / 5) * 5 + (-7 % 5)
   == (  -1  ) * 5 + (-7 % 5)
   ==           -5 + (-7 % 5)
-2 ==                (-7 % 5)        (-7 % 5) is equal to -2
```

Calculating (–7 % –5):

Click here to view code image

```
-7 == (-7 / -5) * -5 + (-7 % -5)
   == (    1    ) * -5 + (-7 % -5)
   ==               -5 + (-7 % -5)
-2 ==                    (-7 % -5)      (-7 % -5) is equal to -2
```

The remainder can be negative only if the dividend is negative, and the sign of the divisor is irrelevant. A shortcut to evaluating the remainder involving negative operands is the following: Ignore the signs of the operands, calculate the remainder, and negate the remainder if the dividend is negative.

Click here to view code image

```
int  r0 =  7  %  7;     //  0
int  r1 =  7  %  5;     //  2
long r2 =  7L % -5L;    //  2L
int  r3 = -7  %  5;     // -2
long r4 = -7L % -5L;    // -2L
boolean relation = -7L == (-7L / -5L) * -5L + r4;  // true
```

An `ArithmeticException` is thrown if the divisor evaluates to zero.

Note that the remainder operator accepts not only integral operands, but also floating-point operands. The *floating-point remainder* `r` is defined by the relation

$$r == a - (b * q)$$

where `a` and `b` are the dividend and the divisor, respectively, and `q` is the *integer* quotient of `(a/b)`. The following examples illustrate a floating-point remainder operation:

Click here to view code image

```
double  dr0 =  7.0  %  7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
boolean fpRelation = dr2  == (-7.0) - (-5.0) * (long)(-7.0 / -5.0);  // true
float   fr3 = -7.0F %  0.0F;   // NaN
```

Additive Binary Operators: +, –

The addition operator `+` and the subtraction operator `-` behave as their names imply: They add and subtract values, respectively. The binary operator `+` also acts as *string concatenation* if any of its operands is a string (p. 67).

Additive operators have lower precedence than all the other arithmetic operators. Table 2.21 includes examples that show how precedence and associativity are used in arithmetic expression evaluation.

Table 2.21 *Examples of Arithmetic Expression Evaluation*

Arithmetic expression	Evaluation	Result when printed
`3 + 2 - 1`	`((3 + 2) - 1)`	`4`
`2 + 6 * 7`	`(2 + (6 * 7))`	`44`
`-5 + 7 - -6`	`(((-5) + 7) - (-6))`	`8`
`2 + 4 / 5`	`(2 + (4 / 5))`	`2`
`13 % 5`	`(13 % 5)`	`3`
`11.5 % 2.5`	`(11.5 % 2.5)`	`1.5`
`10 / 0`		`ArithmeticException`
`2 + 4.0 / 5`	`(2.0 + (4.0 / 5.0))`	`2.8`
`4.0 / 0.0`	`(4.0 / 0.0)`	`Infinity`
`-4.0 / 0.0`	`((-4.0) / 0.0)`	`-Infinity`
`0.0 / 0.0`	`(0.0 / 0.0)`	`NaN`

Numeric Promotions in Arithmetic Expressions

Unary numeric promotion is applied to the single operand of the unary arithmetic operators `-` and `+`. When a unary arithmetic operator is applied to an operand whose type is narrower than `int`, the operand is promoted to a value of type `int`, with the operation resulting in an `int` value. If the conditions for implicit narrowing conversion are not fulfilled (p. 56), assigning the `int` result to a variable of a narrower type will require a cast. This is demonstrated by the following example, where the `byte` operand `b` is promoted to an `int` in the expression `(-b)`:

Click here to view code image

```
byte b = 3;          // int literal in range. Narrowing conversion.
b = (byte) -b;       // Cast required on assignment.
```

Binary numeric promotion is applied to operands of binary arithmetic operators. Its application leads to type promotion for the operands, as explained in §2.4, p. 49. The result is of the promoted type, which is always type `int` or wider. For the expression at (1) in Example 2.2, numeric promotions proceed as shown in Figure 2.4. Note the integer division performed in evaluating the subexpression `(c / s)`.

Figure 2.4 *Numeric Promotion in Arithmetic Expressions*

Example 2.2 *Numeric Promotion in Arithmetic Expressions*

Click here to view code image

```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';                          // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * -b) - (c / s);     // (1) 4888.0D
    System.out.println("Value of v: " + v);
  }
}
```

Output from the program:

```
Value of v: 4888.0
```

In addition to the binary numeric promotions in arithmetic expression evaluation, the resulting value can undergo an implicit widening conversion if assigned to a variable. In the first two declaration statements that follow, only assignment conversions take place. Numeric promotions take place in the evaluation of the right-hand expression in the other declaration statements.

Click here to view code image

```
Byte    b = 10;       // Constant in range: narrowing and boxing on assignment.
Short   s = 20;       // Constant in range: narrowing and boxing on assignment.

char    c = 'z';      // 122 (\u007a)
int     i = s * b;    // Values in s and b promoted to int: unboxing, widening.
long    n = 20L + s;  // Value in s promoted to long: unboxing, widening.
float   r = s + c;    // Value in s is unboxed. This short value and the char
                      // value in c are promoted to int, followed by implicit
                      // widening conversion of int to float on assignment.
double d = r + i;     // Value in i promoted to float, followed by implicit
                      // widening conversion of float to double on assignment.
```

Binary numeric promotion for operands of binary operators implies that each operand of a binary operator is promoted to type `int` or a broader numeric type, if necessary. As with unary operators, care must be exercised in assigning the value resulting from applying a binary operator to operands of these types.

Click here to view code image

```
short h = 40;          // OK: int converted to short. Implicit narrowing.
h = h + 2;             // Error: cannot assign an int to short.
```

The value of the expression `h + 2` is of type `int`. Although the result of the expression is in the range of `short`, this cannot be determined at compile time. The assignment requires a cast.

```
h = (short) (h + 2);   // OK
```

Notice that applying the cast operator `(short)` to the individual operands does not work:

Click here to view code image

```
h = (short) h + (short) 2;     // The resulting value should be cast.
```

Neither does the following approach, which results in a compile-time error:

Click here to view code image

```
h = (short) h + 2;             // The resulting value should be cast.
```

In this case, binary numeric promotion leads to an `int` value as the result of evaluating the expression on the right-hand side, and therefore, requires an additional cast to narrow it to a `short` value.

Arithmetic Compound Assignment Operators: `*=, /=, %=, +=, -=`

A *compound assignment operator* has the following syntax:

variable op`=` *expression*

and the following semantics:

variable `=` `(`*type*`)` `((`*variable*`)` *op* `(`*expression*`))`

The type of the *variable* is *type* and the *variable* is evaluated only once. Note the cast and the parentheses implied in the semantics. Here *op*`=` can be any of the compound assignment operators specified in Table 2.18. The compound assignment operators have the lowest precedence of all the operators in Java, allowing the expression on the right-hand side to be evaluated before the assignment. Table 2.22 defines the arithmetic compound assignment operators.

Table 2.22 *Arithmetic Compound Assignment Operators*

Expression	**Given T as the numeric type of `x`, the expression is evaluated as:**
`x *= a`	`x = (T) ((x) * (a))`
`x /= a`	`x = (T) ((x) / (a))`
`x %= a`	`x = (T) ((x) % (a))`
`x += a`	`x = (T) ((x) + (a))`
`x -= a`	`x = (T) ((x) - (a))`

The implied cast operator, `(T)`, in the compound assignments becomes necessary when the result must be narrowed to the target type. This is illustrated by the following examples:

Click here to view code image

```
int i = 2;
i *= i + 4;             // (1) Evaluated as i = (int) ((i) * (i + 4)).

Integer iRef = 2;
iRef *= iRef + 4;       // (2) Evaluated as iRef = (Integer) ((iRef) * (iRef +

byte b = 2;
b += 10;                // (3) Evaluated as b = (byte) (b + 10).
b = b + 10;             // (4) Will not compile. Cast is required.
```

At (1) the source `int` value is assigned to the target `int` variable, and the cast operator `(int)` in this case is an *identity conversion* (i.e., conversion from a type to the same type). Such casts are permitted. The assignment at (2) entails unboxing to evaluate the expression on the right-hand side, followed by boxing to assign the `int` value. However, at (3), as the source value is an `int` value because the `byte` value in `b` is promoted to `int` to carry out the addition, assigning it to a target `byte` variable requires an implicit narrowing conversion. The situation at (4) with simple assignment will not compile because implicit narrowing conversion is not applicable.

The *variable* is evaluated only once in the expression, not twice, as one might infer from the definition of the compound assignment operator. In the following assignment, `a[i]` is evaluated just once:

Click here to view code image

```
int[] a = new int[] { 2020, 2030, 2040 };
int i = 2;
a[i] += 1;      // Evaluates as a[2] = a[2] + 1, and a[2] gets the value 2041.
```

Implicit narrowing conversions are also applied to increment and decrement operators (p. 69).

Boolean logical compound assignment operators are covered in §2.14, p. 78.

2.9 The Binary String Concatenation Operator `+`

The binary operator `+` is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a `String` object, a string concatenation is performed rather than numeric addition. String concatenation results in a newly created `String` object in which the characters in the text representation of the left-hand operand precede the characters in the text representation of the right-hand operand. It might be necessary to perform a *string conversion* on the non-`String` operand before the string concatenation can be performed. The `String` class is discussed in §8.4, p. 439.

A *string conversion* is performed on the non-`String` operand as follows:

- For an operand of a primitive data type, its value is converted to a text representation.
- For all reference value operands, a text representation is constructed by calling the no-argument `toString()` method on the referred object. Most classes override this method from the `Object` class so as to provide a more meaningful text representation of their objects. Discussion of the `toString()` method can be found in §8.2, p. 425.
- Values like `true`, `false`, and `null` have text representations that correspond to their names. A reference variable with the value `null` also has the text representation `"null"` in this context.

The operator `+` is left associative and has the same precedence level as the additive operators, whether it is performed as a string concatenation or as a numeric addition.

Click here to view code image

```
String strVal = "" + 2020;                      // (1) "2020"
String theName = " Uranium";
theName = " Pure" + theName;                    // (2) " Pure Uranium"
String trademark1 = 100 + "%" + theName;        // (3) "100% Pure Uranium"
```

Since the `+` operator is left-associative, the evaluation at (3) proceeds as follows: The `int` value `100` is concatenated with the string literal `"%"`, followed by concatenation with the contents of the `String` object referred to by the `theName` reference.

Note that using the character literal `'%'` instead of the string literal `"%"` in line (2) does not give the same result:

Click here to view code image

```
String trademark2 = 100 + '%' + theName;        // (4) "137 Pure Uranium"
```

Integer addition is performed by the first `+` operator: `100 + '%';` that is, `(100 + 37)`.

Caution should be exercised because the `+` operator might not be applied as intended, as shown by the following example:

Click here to view code image

```
System.out.println("We can put two and two together and get " + 2 + 2);    // (
```

This statement prints `"We can put two and two together and get 22"`. String concatenation proceeds from left to right: The `String` literal is concatenated with the first `int` literal `2`, followed by concatenation with the second `int` literal `2`. Both occurrences of the `+` operator are treated as string concatenation. To convey the intended meaning of the sentence, parentheses are necessary:

Click here to view code image

```
System.out.println("We can put two and two together and get " + (2 + 2)); // (
```

This statement prints `"We can put two and two together and get 4"`, since the parentheses enforce integer addition in the expression `(2 + 2)` before string concatenation is performed with the contents of the `String` operand.

The following statement will print the correct result, even without the parentheses, because the `*` operator has higher precedence than the `+` operator:

Click here to view code image

```
System.out.println("2 * 2 = " + 2 * 2);       // (7) 2 * 2 = 4
```

Creation of temporary `String` objects might be necessary to store the results of performing successive string concatenations in a `String`-valued expression. For a `String`-valued *constant expression* ((1), (5), (6), and (7) in the preceding examples), the compiler computes such an expression at compile time, and the result is treated as a string literal in the program. The compiler uses a *string builder* to avoid the overhead of temporary `String` objects when applying the string concatenation operator (`+`) in `String`-valued non-constant expressions ((2), (3), and (4) in the preceding examples), as explained in §8.5, p. 469.

2.10 Variable Increment and Decrement Operators: `++`, `--`

Variable increment (`++`) and decrement (`--`) operators come in two flavors: *prefix* and *postfix*. These unary operators have the side effect of changing the value of the arithmetic operand, which must evaluate to a variable. Depending on the operator used, the variable is either incremented by 1 or decremented by 1.

These operators cannot be applied to a variable that is declared `final` and that has been initialized, as the side effect would change the value in such a variable.

These operators are useful for updating variables in loops where only the side effect of the operator is of interest.

The Increment Operator `++`

The prefix increment operator has the following semantics: `++i` adds `1` to the value in `i`, and stores the new value in `i`. It returns the *new* value as the value of the expression. It is equivalent to the following statements:

```
i += 1;
result = i;
return result;
```

The postfix increment operator has the following semantics: `j++` adds `1` to the value in `j`, and stores the new value in `j`. It returns the *original* value that was in `j` as the value of the expression. It is equivalent to the following statements:

```
result = j;
j += 1;
return result;
```

The Decrement Operator `--`

The prefix decrement operator has the following semantics: `--i` subtracts `1` from the value of `i`, and stores the new value in `i`. It returns the *new* value as the value of the expression. It is equivalent to the following statements:

```
i -= 1;
result = i;
return result;
```

The postfix decrement operator has the following semantics: `j--` subtracts `1` from the value of `j`, and stores the new value in `j`. It returns the *original* value that was in `j` as the value of the expression. It is equivalent to the following statements:

```
result = j;
j -= 1;
return result;
```

This behavior of decrement and increment operators applies to any variable whose type is a numeric primitive type or its corresponding numeric wrapper type. Necessary numeric promotions are performed on the value `1` and the value of the variable. Before the new value is assigned to the variable, it is subjected to any narrowing primitive conversion and/or boxing that might be necessary.

Here are some examples that illustrate the behavior of increment and decrement operators:

Click here to view code image

```
// (1) Prefix order: increment/decrement operand before use.
int i = 10;
int k = ++i + --i;  // ((++i) + (--i)). k gets the value 21 and i becomes 10.
--i;                // Only side effect utilized. i is 9. (expression statemen

Integer iRef = 11;  // Boxing on assignment
--iRef;             // Only side effect utilized. iRef refers to an Integer
                    // object with the value 10. (expression statement)
k = ++iRef + --iRef;// ((++iRef) + (--iRef)). k gets the value 21 and
                    // iRef refers to an Integer object with the value 10.
// (2) Postfix order: increment/decrement operand after use.
long j = 10;
long n = j++ + j--; // ((j++) + (j--)). n gets the value 21L and j becomes 10L
j++;                // Only side effect utilized. j is 11L. (expression statem
```

An increment or decrement operator, together with its operand, can be used as an *expression statement* (§3.3, p. 101).

Execution of the assignment in the second declaration statement under (1) proceeds as follows:

Click here to view code image

```
k = ((++i) + (--i))         Operands are evaluated from left to right.
k = ( 11   + (--i))         Side effect: i += 1, i gets the value 11.
k = ( 11   +  10)           Side effect: i -= 1, i gets the value 10.
k = 21
```

Execution of the expression statement `--iRef;` under (1) proceeds as follows:

- The value in the `Integer` object referred to by the reference `iRef` is unboxed, resulting in the `int` value `11`.
- The value `11` is decremented, resulting in the value `10`.

- The value `10` is boxed in an `Integer` object, and this object's reference value is assigned to the reference `iRef`.
- The `int` value `10` of the expression statement is discarded.

Expressions where variables are modified multiple times during the evaluation should be avoided because the order of evaluation is not always immediately apparent.

We cannot associate increment and decrement operators. Given that `a` is a variable, we cannot write `(++(++a))`. The reason is that any operand to `++` must evaluate to a variable, but the evaluation of `(++a)` results in a value.

In the next example, both binary numeric promotion and an implicit narrowing conversion are performed to achieve the side effect of modifying the value of the operand. The `int` value of the expression `(++b)` (i.e., `11`) is assigned to the `int` variable `i`. The side effect of incrementing the value of the `byte` variable `b` requires binary numeric promotion to perform `int` addition, followed by an implicit narrowing conversion of the `int` value to `byte` to assign the value to variable `b`.

Click here to view code image

```
byte b = 10;
int  i = ++b;        // i is 11, and so is b.
```

The following example illustrates applying the increment operator to a floating-point operand. The side effect of the ++ operator is overwritten by the assignment.

Click here to view code image

```
double x = 4.5;
x = x + ++x;         // x gets the value 10.0.
```

Review Questions

2.1 Which of the following is not a legal comment in Java?

Select the one correct answer.

a. `/* // */`
b. `/* */ //`

c. `// /* */`
d. `/* /* */`
e. `/* /* */ */`
f. `// //`

2.2 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Assignment {
  public static void main(String[] args) {
    int a, b, c;
    b = 10;
    a = b = c = 20;
    System.out.println(a);
  }
}
```

Select the one correct answer.

a. The code will fail to compile because the compiler will report that the variable `c` in the multiple assignment statement `a = b = c = 20;` has not been initialized.
b. The code will fail to compile because the multiple assignment statement `a = b = c = 20;` is illegal.
c. The code will compile and print `10` at runtime.
d. The code will compile and print `20` at runtime.

2.3 What will be the result of compiling and running the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    String a, b, c;
    c = new String("mouse");
    a = new String("cat");
    b = a;
    a = new String("dog");
    c = b;

    System.out.println(c);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.
b. The program will print `mouse` at runtime.
c. The program will print `cat` at runtime.
d. The program will print `dog` at runtime.
e. The program will randomly print either `cat` or `dog` at runtime.

2.4 Which of the following expressions evaluate to a floating-point value? Select the three correct answers.

a. `2.0 * 3`
b. `2 * 3`
c. `2/3 + 5/7`
d. `2.4 + 1.6`
e. `0x10 * 1L * 300.0`

2.5 What is the value of the expression `(1 / 2 + 3 / 2 + 0.1)`?

Select the one correct answer.

a. `1`
b. `1.1`
c. `1.6`
d. `2`
e. `2.1`

2.6 What is the value of evaluating the following expression: (- `-1-3 * 10 / 5-1`)?

Select the one correct answer.

a. `-8`
b. `-6`
c. `7`
d. `8`
e. `10`
f. None of the above

2.7 What is the result of compiling and running the following program?

Click here to view code image

```
public class Prog1 {
  public static void main(String[] args) {
    int k = 1;
    int i = ++k + k++ + + k;      // (1)
    System.out.println(i);
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of errors in the expression at (1).
b. The program will print the value `3` at runtime.
c. The program will print the value `4` at runtime.
d. The program will print the value `7` at runtime.

e. The program will print the value `8` at runtime.

2.8 Which is the first line that will cause a compile-time error in the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    char c;
    int i;
    c = 'a'; // (1)
    i = c;   // (2)
    i++;     // (3)
    c = i;   // (4)
    c++;     // (5)
  }
}
```

Select the one correct answer.

a. (1)
b. (2)
c. (3)
d. (4)
e. (5)
f. None of the above. The compiler will not report any errors.

2.9 What will be the result of compiling and running the following program?

Click here to view code image

```
public class EvaluationOrder {
  public static void main(String[] args) {
    int[] array = { 4, 8, 16 };
    int i = 1;
    array[++i] = --i;
    System.out.println(array[0] + array[1] + array[2]);
  }
}
```

Select the one correct answer.

a. `13`
b. `14`
c. `20`
d. `21`
e. `24`

2.11 Boolean Expressions

As the name implies, a boolean expression has the `boolean` data type and can only evaluate to the value `true` or `false`. Boolean expressions, when used as conditionals in control statements, allow the program flow to be controlled during execution.

Boolean expressions can be formed using *relational operators* (p. 74), *equality operators* (p. 75), *boolean logical operators* (p. 78), *conditional operators* (p. 80), the *assignment operator* (p. 54), and the `instanceof` operator (§5.11, p. 269).

2.12 Relational Operators: `<`, `<=`, `>`, `>=`

Given that `a` and `b` represent numeric expressions, the relational (also called *comparison*) operators are defined as shown in Table 2.23.

Table 2.23 *Relational Operators*

`a < b`	`a` less than `b`?
`a <= b`	`a` less than or equal to `b`?
`a > b`	`a` greater than `b`?
`a >= b`	`a` greater than or equal to `b`?

All relational operators are binary operators and their operands are numeric expressions. Binary numeric promotion is applied to the operands of these operators. The evaluation results in a `boolean` value. Relational operators have lower precedence than arithmetic operators, but higher than that of the assignment operators.

Click here to view code image

```
double hours = 45.5;
Double time = 18.0;               // Boxing of double value.
boolean overtime = hours >= 35; // true. Binary numeric promotion: double <--
boolean beforeMidnight = time < 24.0;// true. Unboxing of value in time refere
char letterA = 'A';
boolean order = letterA < 'a';   // true. Binary numeric promotion: int <-- cha
```

Relational operators are nonassociative. Mathematical expressions like $a \leq b \leq c$ must be written using relational and boolean logical/conditional operators.

Click here to view code image

```
int a = 1, b = 7, c = 10;
boolean illegal = a <= b <= c;       // (1) Illegal. Compile-time error!
boolean valid2 = a <= b && b <= c;   // (2) OK.
```

Since relational operators have left associativity, the evaluation of the expression `a <= b <= c` at (1) in these examples would proceed as follows: `((a <= b) <= c)`. Evaluation of `(a <= b)` would yield a `boolean` value that is not permitted as an operand of a relational operator; that is, `(`*boolean value* `<= c)` would be illegal.

2.13 Equality

We distinguish between *primitive data value equality*, *object reference equality*, and *object value equality*.

The equality operators have lower precedence than the relational operators, but higher precedence than the assignment operators.

Primitive Data Value Equality: `==`, `!=`

Given that `a` and `b` represent operands of primitive data types, the primitive data value equality operators are defined as shown in Table 2.24.

Table 2.24 *Primitive Data Value Equality Operators*

`a == b`	Determines whether `a` and `b` are equal—that is, have the same primitive value (*equality*).
`a != b`	Determines whether `a` and `b` are not equal—that is, do not have the same primitive value (*inequality*).

The equality operator `==` and the inequality operator `!=` can be used to compare primitive data values, including `boolean` values. Binary numeric promotion may be applied to the non-boolean operands of these equality operators.

Click here to view code image

```
int year = 2002;
boolean isEven  = year % 2 == 0;      // true.
boolean compare = '1' == 1;           // false. Binary numeric promotion applie
boolean test    = compare == false;   // true.
```

Care must be exercised when comparing floating-point numbers for equality, as an infinite number of floating-point values can be stored only as approximations in a finite number of bits. For example, the expression `(1.0 - 2.0/3.0 == 1.0/3.0)` returns `false`, although mathematically the result should be true.

Analogous to the discussion for relational operators, mathematical expressions like $a = b = c$ must be written using relational and logical/conditional operators. Since equality operators have left associativity, the evaluation of the expression `a==b==c` would proceed as follows: `((a == b) == c)`. Evaluation of `(a == b)` would yield a `boolean` value that *is* permitted as an operand of a data value equality operator, but `(`*boolean value* `== c)` would be illegal if `c` had a numeric type. This problem is illustrated in the following examples. The expression at (1) is illegal, but those at (2) and (3) are legal.

Click here to view code image

```
int a, b, c;
a = b = c = 5;
boolean illegal = a == b == c;              // (1) Illegal.
boolean valid2 = a == b && b == c;          // (2) Legal.
boolean valid3 = a == b == true;            // (3) Legal.
```

Object Reference Equality: `==`, `!=`

The equality operator `==` and the inequality operator `!=` can be applied to reference variables to test whether they refer to the same object. Given that `r` and `s` are reference variables, the reference equality operators are defined as shown in Table 2.25.

Table 2.25 *Reference Equality Operators*

`r == s`	Determines whether `r` and `s` are equal—that is, have the same reference value and therefore refer to the same object (also called *aliases*) (*equality*).
`r != s`	Determines whether `r` and `s` are not equal—that is, do not have the same reference value and therefore refer to different objects (*inequality*).

The operands must be cast compatible: It must be possible to cast the reference value of the one into the other's type; otherwise a compile-time error will result. Casting of references is discussed in §5.8, p. 261.

Click here to view code image

```
Pizza pizzaA = new Pizza("Sweet&Sour");     // new object
Pizza pizzaB = new Pizza("Sweet&Sour");     // new object
Pizza pizzaC = new Pizza("Hot&Spicy");      // new object

String banner = "Come and get it!";         // new object

boolean test  = banner == pizzaA;           // (1) Compile-time error
boolean test1 = pizzaA == pizzaB;           // false
boolean test2 = pizzaA == pizzaC;           // false

pizzaA = pizzaB;                       // Denote the same object; are aliases
boolean test3 = pizzaA == pizzaB;      // true
```

The comparison `banner == pizzaA` at (1) is illegal because the

`String` and `Pizza` types are incompatible operand types as the reference value of one type cannot be cast to the other type. The values of `test1` and `test2` are `false` because the three references denote different objects, regardless of the fact that `pizzaA` and `pizzaB` are both sweet and sour pizzas. The value of `test3` is `true` because now both `pizzaA` and `pizzaB` denote the same object.

The equality and inequality operators are applied to object references to check whether two references denote the same object. The state of the objects that the references denote is not compared. This is the same as testing whether the references are *aliases*, meaning that they denote the same object.

The `null` literal can be assigned to any reference variable, and the reference value in a reference variable can be compared for equality with the `null` literal. The comparison can be used to avoid inadvertent use of a reference variable that does not denote any object.

Click here to view code image

```
if (objRef != null) {
    // ... use objRef ...
}
```

Note that only when the type of *both* operands is either a reference type or the `null` type do these operators test for object reference equality. Otherwise, they test for primitive data equality (see also §8.3, p. 432). In the following code snippet, binary numeric promotion involving unboxing is performed at (1):

Click here to view code image

```
Integer iRef = 10;
boolean b1 = iRef == null;         // Object reference equality
boolean b2 = iRef == 10;           // (1) Primitive data value equality
boolean b3 = null == 10;           // Compile-time error!
```

Object Value Equality

The `Object` class provides the method `public boolean equals(Object obj)`, which can be *overridden* (§5.1, p. 196) to give the right semantics of *object value equality*. The default implementation of this method in the `Object` class returns `true` only if the object is compared with itself, as if the equality operator `==` had been used to compare aliases of an object. Consequently, if a class does not override the semantics of the `equals()` method from the `Object` class, object value equality is the same as object reference equality.

Certain classes in the Java SE API override the `equals()` method, such as `java.lang.String` and the wrapper classes for the primitive data types. For two `String` objects, value equality means they contain identical character sequences. For the wrapper classes, value equality means the wrapper objects have the same primitive value and are of the same wrapper type (see also §8.3, p. 432).

Click here to view code image

```
// Equality for String objects means identical character sequences.
String movie1 = new String("The Revenge of the Exception Handler");
String movie2 = new String("High Noon at the Java Corral");
String movie3 = new String("The Revenge of the Exception Handler");
boolean test0 = movie1.equals(movie2);             // false.
boolean test1 = movie1.equals(movie3);             // true.

// Equality for wrapper classes means same type and same primitive value.
Boolean flag1 = true;                              // Boxing.
Boolean flag2 = false;                             // Boxing.
boolean test2 = flag1.equals("true");              // false. Not same type.
boolean test3 = flag1.equals(!flag2);              // true. Same type and valu

Integer iRef = 100;                                // Boxing.
Short sRef = 100;                                  // Boxing <--- short <--- i
boolean test4 = iRef.equals(100);                  // true. Same type and valu
boolean test5 = iRef.equals(sRef);                 // false. Not same type.
boolean test6 = iRef.equals(3.14);                 // false. Not same type.

// The Pizza class does not override the equals() method, so we can use either
// equals() method inherited from the Object class or equality operator ==.
Pizza pizza1 = new Pizza("Veggies Delight");
Pizza pizza2 = new Pizza("Veggies Delight");
Pizza pizza3 = new Pizza("Cheese Delight");
boolean test7 = pizza1.equals(pizza2);             // false.
boolean test8 = pizza1.equals(pizza3);             // false.
boolean test9 = pizza1 == pizza2;                  // false.
pizza1 = pizza2;                                   // Creates aliases.
boolean test10 = pizza1.equals(pizza2);            // true.
boolean test11 = pizza1 == pizza2;                 // true.
```

2.14 Boolean Logical Operators: `!`, `^`, `&`, `|`

Boolean logical operators include the unary operator `!` (*logical complement*) and the binary operators `&` (*logical* AND), `|` (*logical inclusive* OR), and `^` (*logical exclusive* OR, also called *logical* XOR). These operators can be applied to `boolean` or `Boolean` operands, returning a `boolean` value. The operators `&`, `|`, and `^` can also be applied to integral operands to perform *bitwise* logical operations (p. 82).

Given that *x* and *y* represent boolean expressions, the boolean logical operators are defined in Table 2.26. The precedence of the operators decreases from left to right in the table.

Table 2.26 *Truth Values for Boolean Logical Operators*

Complement
x y
true
true
false
false

These operators always evaluate *both* of the operands, unlike their counterpart *conditional operators* `&&` and `||` (p. 80). Unboxing is applied to the operand values, if necessary. Truth values for boolean logical operators are shown in Table 2.26, where `x` and `y` are either of type `boolean` or `Boolean`.

Operand Evaluation for Boolean Logical Operators

In the evaluation of boolean expressions involving boolean logical AND, XOR, and OR operators, both the operands are evaluated. The order of operand evaluation is always from left to right.

Click here to view code image

```
if (i > 0 & i++ < 10) {/*...*/} // i will be incremented, regardless of value
```

The binary boolean logical operators have lower precedence than the arithmetic and relational operators, but higher precedence than the assignment, conditional AND, and OR operators (p. 80). This is illustrated in the following examples:

Click here to view code image

```
boolean b1, b2, b3 = false, b4 = false;
Boolean b5 = true;
b1 = 4 == 2 & 1 < 4;          // false, evaluated as (b1 = ((4 == 2) & (1 < 4
b2 = b1 | !(2.5 >= 8);        // true
b3 = b3 ^ b5;                 // true, unboxing conversion on b5
b4 = b4 | b1 & b2;            // false
```

Here, the order of evaluation is illustrated for the last expression statement:

Click here to view code image

```
     (b4 = (b4 | (b1 & b2)))
⇒ (b4 = (false | (b1 & b2)))
⇒ (b4 = (false | (false & b2)))
⇒ (b4 = (false | (false & true)))
⇒ (b4 = (false | false))
⇒ (b4 = false)
⇒ false
```

Note that `b2` was evaluated, although strictly speaking, it was not necessary. This behavior is guaranteed for boolean logical operators.

Boolean Logical Compound Assignment Operators: `&=`, `^=`, `|=`

Compound assignment operators for the boolean logical operators are defined in Table 2.27. The left-hand operand must be a boolean variable, and the right-hand operand must be a boolean expression. An identity conversion is applied implicitly on assignment. These operators can also be applied to integral operands to perform *bitwise* compound assignments (p. 82). See also the discussion on arithmetic compound assignment operators (p. 66).

Table 2.27 *Boolean Logical Compound Assignment Operators*

Expression	**Given a and `b` are of type `boolean` or `Boolean`, the expression is evaluated as:**
`b &= a`	`b = (b & (a))`
`b ^= a`	`b = (b ^ (a))`
`b |= a`	`b = (b | (a))`

Here are some examples to illustrate the behavior of boolean logical compound assignment operators:

Click here to view code image

```
boolean b1 = false, b2 = true, b3 = false;
Boolean b4 = false;
b1 |= true;            // true
b4 ^= b1;              // (1) true, unboxing in (b4 ^ (b1)), boxing on assign
b3 &= b1 | b2;         // (2) false, b3 = (b3 & (b1 | b2))
b3 = b3 & b1 | b2;     // (3) true,  b3 = ((b3 & b1) | b2)
```

The assignment at (1) entails unboxing to evaluate the expression on the right-hand side, followed by boxing to assign the `boolean` result. It is also instructive to compare how the assignments at (2) and (3) are performed, as they lead to different results with the same values of the operands, showing how the precedence affects the evaluation.

2.15 Conditional Operators: `&&`, `||`

The conditional operators `&&` and `||` are similar to their counterpart logical operators & and |, except that their evaluation is *short-circuited*. Given that `x` and `y` represent values of `boolean` or `Boolean` expressions, the conditional operators are defined in Table 2.28. In the table, the operators are listed in decreasing order of precedence.

Table 2.28 *Conditional Operators*

`x && y`	Conditional AND	`true` if both operands are true; otherwise `false`.
`x || y`	Conditional OR	`true` if either or both operands are true; otherwise `false`.

Unlike their logical counterparts `&` and `|`, which can also be applied to integral operands for bitwise operations, the conditional operators `&&` and `||` can be applied to only `boolean` operands. Their evaluation results in a `boolean` value. Truth values for conditional operators are shown in Table 2.29. Not surprisingly, the conditional operators have the same truth values as their counterpart logical operators. However, unlike with their logical counterparts, there are no compound assignment operators for the conditional operators.

Table 2.29 *Truth Values for Conditional Operators*

`x`	`y`	AND `x && y`	OR `x || y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Short-Circuit Evaluation

In the evaluation of boolean expressions involving conditional AND and OR, the left-hand operand is evaluated before the right-hand operand, and the evaluation is short-circuited (i.e., if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated). In other words, the right-hand operand is evaluated conditionally.

The binary conditional operators have lower precedence than the arithmetic, relational, and logical operators, but higher precedence than the assignment operators.

Unboxing of the operand value takes place when necessary, before the operation is performed. The following examples illustrate usage of conditional operators:

Click here to view code image

```
Boolean b1 = 4 == 2 && 1 < 4;   // false, short-circuit evaluated as
                                // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;    // true, short-circuit evaluated as
                                // (b2 = ((!b1) || (2.5 > 8)))
Boolean b3 = !(b1 && b2);       // true
boolean b4 = b1 || !b3 && b2;   // false, short-circuit evaluated as
                                // (b4 = (b1 || ((!b3) && b2)))
```

The order of evaluation for computing the value stored in the boolean variable `b4` proceeds as follows:

Click here to view code image

```
     (b4 = (b1 || ((!b3) && b2)))
⇒ (b4 = (false || ((!b3) && b2)))
⇒ (b4 = (false || ((!true) && b2)))
⇒ (b4 = (false || ((false) && b2)))
⇒ (b4 = (false || false))
⇒ (b4 = false)
```

Note that `b2` is not evaluated, short-circuiting the evaluation. Example 2.3 illustrates the short-circuit evaluation of the initialization expressions in the declaration statements given in the code snippet above. In addition, it shows an evaluation (see the declaration of `b5`) involving boolean logical operators that always evaluate both operands. The output shows how many operands were evaluated for each expression. See also Example 2.1, p. 53, which uses a similar approach to illustrate the order of operand evaluation in arithmetic expressions.

Example 2.3 *Short-Circuit Evaluation Involving Conditional Operators*

Click here to view code image

```
public class ShortCircuit {
  public static void main(String[] args) {
    // Boolean b1 = 4 == 2 && 1 < 4;
    Boolean b1 = operandEval(1, 4 == 2) && operandEval(2, 1 < 4);
    System.out.println("Value of b1: " + b1);

    // boolean b2 = !b1 || 2.5 > 8;
    boolean b2 = !operandEval(1, b1) || operandEval(2, 2.5 > 8);
    System.out.println("Value of b2: " + b2);

    // Boolean b3 = !(b1 && b2);
    Boolean b3 = !(operandEval(1, b1) && operandEval(2, b2));
    System.out.println("Value of b3: " + b3);

    // boolean b4 = b1 || !b3 && b2;
    boolean b4 = operandEval(1, b1) || !operandEval(2, b3) && operandEval(3, b
    System.out.println("Value of b4: " + b4);

    // boolean b5 = b1 | !b3 & b2;     // Using boolean logical operators
    boolean b5 = operandEval(1, b1) | !operandEval(2, b3) & operandEval(3, b2)
    System.out.println("Value of b5: " + b5);
  }

  static boolean operandEval(int opNum, boolean operand) {                // (
    System.out.println(opNum);
    return operand;
  }
}
```

Output from the program:

Click here to view code image

```
1
Value of b1: false
1
Value of b2: true
1
Value of b3: true
1
2
Value of b4: false
1
2
3
Value of b5: false
```

Short-circuit evaluation can be used to ensure that a reference variable denotes an object before it is used.

Click here to view code image

```
if (objRef != null && objRef.equals(other)) { /*...*/ }
```

The method call is now conditionally dependent on the left-hand operand and will not be executed if the variable `objRef` has the null reference. If we use the logical `&` operator and the variable `objRef` has the null reference, evaluation of the right-hand operand will result in a `NullPointerException`.

In summary, we employ the conditional operators `&&` and `||` if the evaluation of the right-hand operand is conditionally dependent on the left-hand operand. We use the boolean logical operators `&` and `|` if both operands must be evaluated. The subtlety of conditional operators is illustrated by the following examples:

Click here to view code image

```
if (i > 0 && i++ < 10) {/*...*/}   // i is not incremented if i > 0 is false.
if (i > 0 || i++ < 10) {/*...*/}   // i is not incremented if i > 0 is true.
```

2.16 Integer Bitwise Operators: `~`, `&`, `|`, `^`

A review of integer representation (p. 33) is recommended before continuing with this section on how integer bitwise operators can be applied to values of *integral* data types.

Integer bitwise operators include the unary operator `~` (*bitwise complement*) and the binary operators `&` (*bitwise* AND), `|` (*bitwise inclusive* OR), and `^` (*bitwise exclusive* OR, also known as *bitwise* XOR). The operators `&`, `|`, and `^` are overloaded, as they can be applied to `boolean` or `Boolean` operands to perform *boolean* logical operations (p. 78).

The binary bitwise operators perform bitwise operations between corresponding individual bit values in the operands. Unary numeric promotion is applied to the operand of the unary bitwise complement operator `~`, and binary numeric promotion is applied to the operands of the binary bitwise operators. The result is a new integer value of the promoted type, which can be either `int` or `long`.

Given that `A` and `B` are corresponding bit values (either `0` or `1`) in the left-hand and right-hand operands, respectively, these bitwise operators are defined as shown in Table 2.30. The operators are listed in order of decreasing precedence.

Table 2.30 *Integer Bitwise Operators*

Operator		Effect on each bit of the binary representation
Bitwise complement	`~A`	Invert the bit value: `1` to `0`, `0` to `1`.
Bitwise AND	`A & B`	`1` if both bits are `1`; otherwise `0`.
Bitwise OR	`A | B`	`1` if either or both bits are `1`; otherwise `0`.
Bitwise XOR	`A ^ B`	`1` if and only if one of the bits is `1`; otherwise `0`.

The result of applying bitwise operators between two corresponding bits in the operands is shown in Table 2.31, where `A` and `B` are corresponding bit values in the left-hand and right-hand operands, respectively. Table 2.31 is analogous to Table 2.26 for boolean logical operators, if we consider bit value `1` to represent `true` and bit value `0` to represent `false`.

Table 2.31 *Result Table for Bitwise Operators*

		Complement	AND	OR	XOR
A	**B**	**~A**	**A & B**	**A | B**	**A ^ B**
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Examples of Bitwise Operator Application

Click here to view code image

```
char v1 = ')';           // Unicode value 41
byte v2 = 13;
int result1 = ~v1;       // -42
int result2 = v1 & v2;   // 9
int result3 = v1 | v2;   // 45
int result4 = v1 ^ v2;   // 36
```

Table 2.32 shows how the result is calculated. Unary and binary numeric promotions are applied first, converting the operands to `int` in these cases. Note that the operator semantics are applied to corresponding individual bits—that is, the first bit of the left-hand operand and the first bit of the right-hand operand, the second bit of the left-hand operand and the second bit of the right-hand operand, and so on.

Table 2.32 *Examples of Bitwise Operations*

~v1	v1 & v2	v1 | v2	v1 ^ v2
~ 0...0010 1001	0...0010 1001	0...0010 1001	0...0010 1001
	& 0...0000 1101	| 0...0000 1101	^ 0...0000 1101
= 1...1101 0110	= 0...0000 1001	= 0...0010 1101	= 0...0010 0100
= 0xffffffd6	= 0x00000009	= 0x0000002d	= 0x00000024
= -42	= 9	= 45	= 36

It is instructive to run examples and print the result of a bitwise operation in different notations, as shown in Example 2.4. The integer bitwise operators support a programming technique called *bit masking*. The value `v2` is usually called a *bit mask*. Depending on the bitwise operation performed on the value `v1` and the mask `v2`, we see how the resulting value reflects the bitwise operation performed between the individual corresponding bits of the value `v1` and the mask `v2`. By choosing appropriate values for the bits in the mask `v2` and the right bitwise operation, it is possible to extract, set, and toggle specific bits in the value `v1`.

Methods for converting integers to strings in different notations can be found in the `Integer` class (§8.3, p. 435).

Example 2.4 *Bitwise Operations*

Click here to view code image

```
public class BitOperations {
  public static void main(String[] args) {
    char v1 = ')';                        // Unicode value 41
    byte v2 = 13;
    printIntToStr("v1:", v1);             // 41
    printIntToStr("v2:", v2);             // 13
    printIntToStr("~v1:", ~v1);           // -42
```

```
      printIntToStr("v1 & v2:", v1 & v2); // 9
      printIntToStr("v1 | v2:", v1 | v2); // 45
      printIntToStr("v1 ^ v2:", v1 ^ v2); // 36
    }

    public static void printIntToStr(String label, int result) {
      System.out.println(label);
      System.out.println("    Binary:  " + Integer.toBinaryString(result));
      System.out.println("    Hex:     " + Integer.toHexString(result));
      System.out.println("    Decimal: " + result);
    }
  }
```

Output from the program:

Click here to view code image

```
v1:
    Binary: 101001
    Hex:     29
    Decimal: 41
v2:
    Binary:  1101
    Hex:     d
    Decimal: 13
~v1:
    Binary:  11111111111111111111111111010110
    Hex:     ffffffd6
    Decimal: -42
v1 & v2:
    Binary:  1001
    Hex:     9
    Decimal: 9
v1 | v2:
    Binary:  101101
    Hex:     2d
    Decimal: 45
v1 ^ v2:
    Binary:  100100
    Hex:     24
    Decimal: 36
```

Bitwise Compound Assignment Operators: `&=`, `^=`, `|=`

Bitwise compound assignment operators for the bitwise operators are defined in Table 2.33. Type conversions for these operators, when applied to integral operands, are the same as for other compound assignment operators: An implicit narrowing conversion is performed on assignment when the destination data type is either `byte`, `short`, or `char`. These operators can also be applied to boolean operands to perform logical compound assignments (p. 79).

Table 2.33 *Bitwise Compound Assignment Operators*

Expression:	**Given T is the integral type of `b`, the expression is evaluated as:**
`b &= a`	`(T) ((b) & (a))`
`b ^= a`	`(T) ((b) ^ (a))`

```
b |=(Ta) ((b) | (a))
```

Examples of Bitwise Compound Assignment

Click here to view code image

```
int  v0 = -42;
char v1 = ')';   // 41
byte v2 = 13;
v0 &= 15;        //     1...1101 0110 & 0...0000 1111 => 0...0000 0110 (= 6)
v1 |= v2;        // (1) 0...0010 1001 | 0...0000 1101 => 0...0010 1101 (= 45, '
```

At (1) in these examples, both the `char` value in `v1` and the `byte` value in `v2` are first promoted to `int`. The result is implicitly narrowed to the destination type `char` on assignment.

2.17 Shift Operators: <<, >>, >>>

The binary shift operators return a new value formed by shifting bits either left or right a specified number of times in a given integral value. The number of shifts (also called the *shift distance*) is given by the right-hand operand, and the value that is to be shifted is given by the left-hand operand. Note that *unary* numeric promotion is applied to each operand *individually*. The value returned has the promoted type of the left-hand operand. Also, the value of the left-hand operand is *not* affected by applying the shift operator.

The shift distance is calculated by AND-ing the value of the right-hand operand with a mask value of `0x1f` (31) if the left-hand operand has the promoted type `int`, or using a mask value of `0x3f` (63) if the left-hand operand has the promoted type `long`. This effectively means masking the five lower bits of the right-hand operand in the case of an `int` left-hand operand, and masking the six lower bits of the right-hand operand in the case of a `long` left-hand operand. Thus the shift distance is always in the range 0 to 31 when the promoted type of the left-hand operand is `int` (which has size 32 bits), and in the range 0 to 63 when the promoted type of the left-hand operand is `long` (which has size 64 bits).

Given that `v` contains the value whose bits are to be shifted and `n` specifies the number of bits to shift, the bitwise operators are defined in Table 2.34. It is implied that the value `n` in Table 2.34 is subject to the shift distance calculation outlined above, and that the shift operations are always performed on the value of the left-hand operand represented in two’s complement.

Table 2.34 *Shift Operators*

`v << n`	Shift left	Shift all bits in `v` left `n` times, filling with `0` from the right.
`v >> n`	Shift right with sign bit	Shift all bits in `v` right `n` times, filling with the sign bit from the left.
`v >>> n`	Shift right with zero fill	Shift all bits in `v` right `n` times, filling with `0` from the left.

Since `char`, `byte`, and `short` operands are promoted to `int`, the result of applying these bitwise operators is always either an `int` or a `long` value. Care must be taken in employing a cast to narrow the resulting value, as this can result in a loss of information as the upper bits are discarded during conversion.

Note that regardless of the promotion of the values in the operands or determination of the shift distance, the operands `v` and `n` are not affected by these three shift operators. However, the shift compound assignment operators, discussed in this section, can change the value of the left-hand operand `v`.

Bit values shifted out (*falling off*) from bit `0` or the most significant bit are lost. Since bits can be shifted both left and right, a positive value when shifted can result in a negative value, and vice versa.

The Shift-Left Operator `<<`

As the bits are shifted left, zeros are always filled in from the right.

Click here to view code image

```
int i = 12;
int result = i << 4;    // 192
```

The bits in the `int` value for `i` are shifted left four places as follows:

Click here to view code image

```
i << 4
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

Each left-shift corresponds to multiplication of the value by 2. In the above example, `12 * 2`^{4} is `192`.

The sign bit of a `byte` or `short` value is extended to fill the higher bits when the value is promoted, as illustrated by the example below:

Click here to view code image

```
byte  b = -42;          // 11010110
short n = 4;
int   result = b << n;  // -672
```

The values of the two operands, `b` and `n`, in the previous example are promoted individually. The `short` value in `n` is promoted to `int`. The `byte` value in `b`, after promotion to `int`, is shifted left `4` places:

Click here to view code image

```
b << n
= 1101 0110 << 0000 0000 0000 0100
= 1111 1111 1111 1111 1111 1111 1101 0110 <<0000 0000 0000 0000 0000 0000 0000
= 1111 1111 1111 1111 1111 1111 1101 0110 << 4
= 1111 1111 1111 1111 1111 1101 0110 0000
= 0xfffffd60
= -672
```

In the above example, `-42 * 2`^{4} is `-672`.

Care must also be taken when assigning the result of a shift operator to a narrower data type.

Click here to view code image

```
byte a = 32;
int  j = a << 3;            // 256
byte b = (byte) (a << 3);   // 0. Cast mandatory.
```

The result of `(a << 3)` is `256`.

Click here to view code image

```
a << 3
= 0000 0000 0000 0000 0000 0000 0010 0000 << 3
= 0000 0000 0000 0000 0000 0001 0000 0000
= 0x00000100
= 256
```

The value that `j` gets is `256`, but the value that `b` gets is `0`, as the higher bits are discarded in the explicit narrowing conversion.

The examples above do not show how the shift distance is determined. It is obvious from the value of the right-hand operand, which is within the range 0 to 31, inclusive. An example with the shift-left operator, where the value of the right-hand operand is out of range, is shown below.

Click here to view code image

```
12 << 36
```

```
= 0000 0000 0000 0000 0000 0000 0000 1100 << (0...0010 0100 & 0001 1111)
= 0000 0000 0000 0000 0000 0000 0000 1100 << 0...0000 0100
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

The value of the right-hand operand, `36`, is AND-ed with the mask `11111` (i.e., `31`, `0x1f`), giving the shift distance `4`. This is the same as `(36 % 32)`. It is not surprising that `(12 << 36)` is equal to `(12 << 4)` (i.e., `192`).

The Shift-Right-with-Sign-Fill Operator `>>`

As the bits are shifted right, the sign bit (the most significant bit) is used to fill in from the left. So, if the left-hand operand is a positive value, zeros are filled in from the left, but if the operand is a negative value, ones are filled in from the left.

Click here to view code image

```
int i = 12;
int result = i >> 2;    // 3
```

The value for `i` is shifted right with sign-fill two places.

Click here to view code image

```
i >> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

Each right-shift corresponds to integer division of the value by 2, but this can give unexpected results if care is not exercised, as bits start falling off. In the above example, `12 / 2`^{2} is `3`.

Similarly, when a negative value is shifted right, ones are filled in from the left.

Click here to view code image

```
byte b = -42;           // 11010110
int result = b >> 4;    // -3
```

The `byte` value for `b`, after promotion to `int`, is shifted right with sign-fill four places.

Click here to view code image

```
b >> 4
```

```
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 4
= 1111 1111 1111 1111 1111 1111 1111 1101
= 0xfffffffd
= -3
```

In the following example, the right-hand operand has a negative value:

Click here to view code image

```
-42 >> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >> (1...1111 1100 & 0001 1111)
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 0...0001 1100
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 28
= 1111 1111 1111 1111 1111 1111 1111 1111
= 0xffffffff
= -1
```

The value of the right-hand operand, `-4`, is AND-ed with the mask `11111` (i.e., `31`, `0x1f`), giving the shift distance `28`. This is the same as `(-4 % 32)`. The value of `(-42 >> -4)` is equivalent to `(-42 >> 28)`.

The Shift-Right-with-Zero-Fill Operator `>>>`

As the bits are shifted right, zeros are filled in from the left, regardless of whether the operand has a positive or a negative value.

Obviously, for positive values, the shift-right-with-zero-fill `>>>` and shift-right-with-sign-fill `>>` operators are equivalent. The expression `(12 >> 2)` and the expression `(12 >>> 2)` return the same value:

Click here to view code image

```
12 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

Individual unary numeric promotion of the left-hand operand is shown in the following example:

Click here to view code image

```
byte b = -42;              // 1101 0110
int result = b >>> 4;      // 268435453
```

It is instructive to compare the value of the expression `(-42 >>> 4)` with that of the expression `(-42 >> 4)`, which has the value `-3`. The `byte` value for `b`, after unary numeric promotion to `int`, is shifted right with zero-fill four places.

Click here to view code image

```
b >>> 4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 4
= 0000 1111 1111 1111 1111 1111 1111 1101
= 0x0ffffffd
= 268435453
```

In the following example, the value of the right-hand operand is out of range, resulting in a shift distance of `28` (as we have seen before):

Click here to view code image

```
-42 >>> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 28
= 0000 0000 0000 0000 0000 0000 0000 1111
= 0x0000000f
= 15
```

Shift Compound Assignment Operators: `<<=`, `>>=`, `>>>=`

Table 2.35 lists shift compound assignment operators. Type conversions for these operators, when applied to integral operands, are the same as for other compound assignment operators: An implicit narrowing conversion is performed on assignment when the destination data type is either `byte`, `short`, or `char`.

Table 2.35 *Shift Compound Assignment Operators*

Expression	Given T as the integral type of `v`, the expression is evaluated as:
`v <<= n`	`v = (T) ((v) << (n))`
`v >>= n`	`v = (T) ((v) >> (n))`
`v >>>= n`	`v = (T) ((v) >>> (n))`

Examples of Shift Compound Assignment Operators

Click here to view code image

```
int i = -42;
i >>= 4;                    // 1...1101 0110 >> 4 => 1...1111 1101 (= -3).
byte a = 12;
a <<= 5;                    // (1) -128. Evaluated as a = (byte)((int)a << 5)
a = a << 5;                 // Compile-time error! Needs explicit cast.
```

The example at (1) illustrates the truncation that takes place on narrowing to the destination type. The `byte` value in `a` is first promoted to `int` (by applying unary numeric promotion in this case), then shifted left five places, followed by implicit narrowing to `byte`:

Click here to view code image

```
a = (byte) (a << 5)
  = (byte) (0000 0000 0000 0000 0000 0000 0000 1100 << 5)
  = (byte)  0000 0000 0000 0000 0000 0001 1000 0000
  = 1000 0000
  = 0x80
  = -128
```

2.18 The Conditional Operator `?:`

The ternary conditional operator `?:` allows *conditional expressions* to be defined. The conditional expression has the following syntax:

condition ? $expression_1$: $expression_2$

It is called ternary because it has three operands. If the boolean expression *condition* is `true`, then $expression_1$ is evaluated; otherwise, $expression_2$ is evaluated. Both $expression_1$ and $expression_2$ must evaluate to values that can be converted to the *type* of the conditional expression. This type is determined from the types of the two expressions. The value of the evaluated expression is converted to the type of the conditional expression, and may involve autoboxing and unboxing.

Evaluation of a conditional expression is an example of short-circuit evaluation. As only one of the two expressions is evaluated, one should be wary of any side effects in a conditional expression.

In the following code snippet at (1), both expressions in the conditional expression are of type `byte`. The type of the conditional expression is therefore `byte`. That a value of type `byte` can be converted to an `int` by an implicit widening numeric conversion to be assignment compatible with the `int` variable `daysInFebruary` is secondary in determining the type of the conditional expression. Note that the conditional operator at (1) has higher precedence than the assignment operator `=`, making it unnecessary to enclose the conditional expression in parentheses.

Click here to view code image

```
boolean leapYear = false;
byte v29 = 29;
byte v28 = 28;
int daysInFebruary = leapYear ? v29 : v28;   // (1)
```

The following examples illustrate the use of conditional expressions. The type of the conditional expression at (2) is `int`, and no conversion of any expression value is necessary. The type of the conditional expression at (3) is `double`, due to binary numeric promotion: The `int` value of the first expression is promoted to a `double`. The compiler reports an error because a `double` cannot be assigned to an `int` variable. The type of the conditional expression at (4) is also `double` as at (3), but now the `double` value is assignment compatible with the `double` variable `minDoubleValue`.

Click here to view code image

```
int i = 3;
int j = 4;
int minValue1 = i < j ? i : j;                        // (2) int
int minValue2 = i < j ? i : Double.MIN_VALUE;         // (3) double. Not OK.
double minDoubleValue = i < j ? i : Double.MIN_VALUE; // (4) double
```

At (5) below, the primitive values of the expressions can be boxed and assigned to an `Object` reference. At (6), the `int` value of the first expression can be boxed in an `Integer`. The `println()` method creates and prints a text representation of any object whose reference value is passed as a parameter.

Click here to view code image

```
// Assume i and j are of type int and initialized correctly.
Object obj = i < j ? i : true;         // (5) value of i boxed in Integer or
                                       //     literal true boxed in Boolean
System.out.println(i < j ? i : "Hi"); // (6) value of i boxed in Integer or
                                       //     String object "Hi"
```

The conditional expression is *not* an expression statement. The following code will not compile:

Click here to view code image

```
(i < j) ? i : j;    // Compile-time error!
```

The conditional expression can be nested, and the conditional operator associates from right to left.

Click here to view code image

```
a?b:c?d:e?f:g evaluates as (a?b:(c?d:(e?f:g)))
```

The value of this conditional expression is `g` if, and only if, `a`, `c`, and `e` are `false`. A nested conditional expression is used in the next example. As a convention, the condition in a conditional expression is enclosed in parentheses to aid in reading the code. Typically, a conditional expression is used when it makes the code easier to read, especially when the expressions are short and without side effects.

Click here to view code image

```
int n = 3;
String msg = (n==0) ? "no cookies." : (n==1) ? "one cookie." : "many cookies."
System.out.println("You get " + msg); // You get many cookies.
```

The conditional operator is the expression equivalent of the `if-else` statement (§4.1, p. 153).

2.19 Other Operators: `new`, `[]`, `instanceof`, `->`

The `new` operator is used to create objects, such as instances of classes and arrays. It is used with a constructor call to instantiate classes (§5.11, p. 269) and with the `[]` notation to create arrays (§3.9, p. 119). It is also used to instantiate anonymous arrays (§3.9, p. 122).

Click here to view code image

```
Pizza onePizza = new Pizza();       // Create an instance of the Pizza class.
```

The `[]` notation is used to declare and construct arrays, and is also used to access array elements (§3.9, p. 120).

Click here to view code image

```
int[] anArray = new int[5];// Declare and construct an int array of 5 elements
anArray[4] = anArray[3];   // Element at index 4 gets value of element at inde
```

The boolean, binary, and infix operator `instanceof` is used for either type comparison or pattern matching (§5.11, p. 269).

Click here to view code image

```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza; // true.
boolean test2 = "Pizza" instanceof Pizza; // Compile-time error. Incompatible
                                          // operand types.
boolean test3 = null instanceof Pizza; // Always false. null is not an instanc
```

The arrow operator `->` is used in a `switch` statement (§4.2, p. 155), in a `switch` expression (§4.3, p. 164), and in the definition of a lambda expression (§13.2, p. 679).

Click here to view code image

```
java.util.function.Predicate<String> predicate = str -> str.length() % 2 == 0;
boolean test4 = predicate.test("The lambda strikes back!");    // true.
```

Review Questions

2.10 Which of the following statements are true? Select the two correct answers.

- **a.** The remainder operator `%` can be used only with integral operands.
- **b.** Short-circuit evaluation occurs with boolean logical operators.
- **c.** The arithmetic operators `*`, `/`, and `%` have the same level of precedence.
- **d.** A `short` value ranges from `-128` to `+127`, inclusive.
- **e.** `(+15)` is a legal expression.

2.11 Which of the following statements are true about the lines of output printed by the following program?

Click here to view code image

```
public class BoolOp {
  static void op(boolean a, boolean b) {
    boolean c = a != b;
    boolean d = a ^ b;
    boolean e = c == d;
    System.out.println(e);
  }

  public static void main(String[] args) {
    op(false, false);
    op(true, false);
    op(false, true);
    op(true, true);
  }
}
```

Select the three correct answers.

- **a.** All lines printed are the same.
- **b.** At least one line contains `false`.
- **c.** At least one line contains `true`.
- **d.** The first line contains `false`.
- **e.** The last line contains `true`.

2.12 What is the result of running the following program?

Click here to view code image

```
public class OperandOrder {
  public static void main(String[] args) {
    int i = 0;
    int[] a = {3, 6};
    a[i] = i = 9;
    System.out.println(i + " " + a[0] + " " + a[1]);
  }
}
```

Select the one correct answer.

a. When run, the program throws an `ArrayIndexOutOfBoundsException`.

b. When run, the program will print `9 9 6`.

c. When run, the program will print `9 0 6`.

d. When run, the program will print `9 3 6`.

e. When run, the program will print `9 3 9`.

2.13 Which of the following statements are true about the output from the following program?

Click here to view code image

```
public class Logic {
  public static void main(String[] args) {
    int i = 0;
    int j = 0;

    boolean t = true;
    boolean r;

    r = (t &  0 < (i+=1));
    r = (t && 0 < (i+=2));
    r = (t |  0 < (j+=1));
    r = (t || 0 < (j+=2));
    System.out.println(i + " " + j);
  }
}
```

Select the two correct answers.

a. The first digit printed is 1.

b. The first digit printed is 2.

c. The first digit printed is 3.

d. The second digit printed is 1.

e. The second digit printed is 2.

f. The second digit printed is 3.

2.14 Given the following code:

Click here to view code image

```
int x = 1, y = 2, z = 3;
if (x < y || ++z > 4) {
    System.out.println("a" + x + y + z);
  }
  if (x < y && ++z > 4) {
    System.out.println("b" + x + y + z);
  }
```

What will be the output?

Select the one correct answer.

a. `a124`
`b125`

b. `a123`

c. `a123`
`b124`

2.15 Which of the following statements when inserted at (1) will not result in a compile-time error?

Click here to view code image

```
public class RQ05A200 {
  public static void main(String[] args) {
    int i = 20;
    int j = 30;
    // (1) INSERT STATEMENT HERE
  }
}
```

Select the three correct answers.

a. `int result1 = i < j ? i : j * 10d;`

b. `int result2 = i < j ? { ++i } : { ++j };`

c. `Number number = i < j ? i : j * 10D;`

d. `System.out.println(i < j ? i);`

e. `System.out.println(i < j ? ++i : ++j);`

f. `System.out.println(i == j ? i == j : "i not equal to j");`

2.16 Which of the following statements are true about the following code?

Click here to view code image

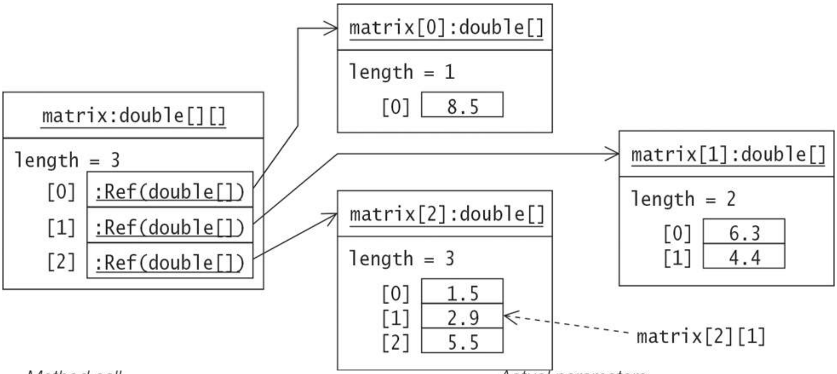
```
public class RQ05A100 {
  public static void main(String[] args) {
    int n1 = 10, n2 = 10;
    int m1 = 20, m2 = 30;
    int result = n1 != n2? n1 : m1 != m2? m1 : m2;
    System.out.println(result);
  }
}
```

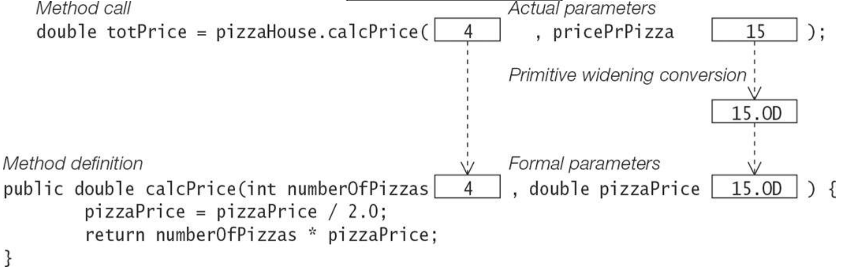
Select the one correct answer.

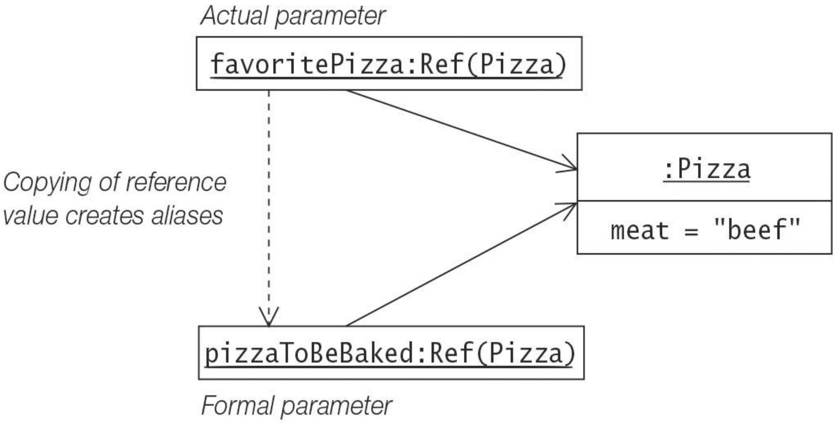
a. The program will fail to compile.

b. The program will throw an `ArithmeticException` at runtime.

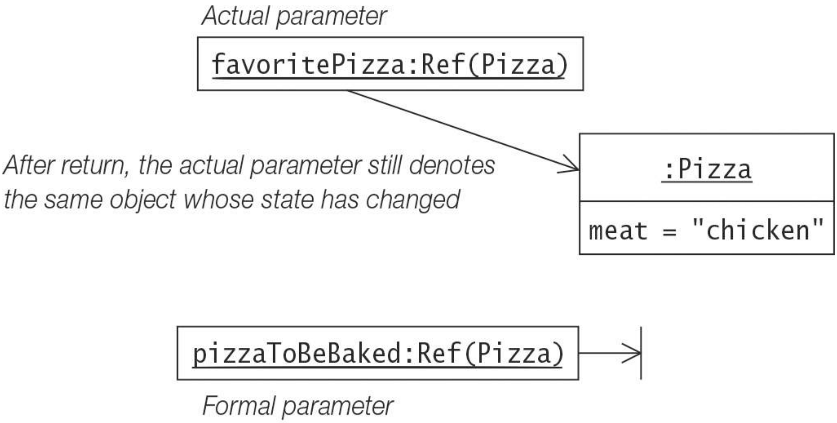
c. The program will compile and print `10` when run.
d. The program will compile and print `20` when run.
e. The program will compile and print `30` when run.


matrix:double[][]
length = 3
[0] :Ref(double[])
[1] :Ref(double[])
[2] :Ref(double[])
matrix[0]:double[]
length = 1
[0] 8.5
matrix[1]:double[]
length = 2
[0] 6.3
[1] 4.4
matrix[2]:double[]
length = 3
[0] 1.5
[1] 2.9
[2] 5.5
matrix[2][1]

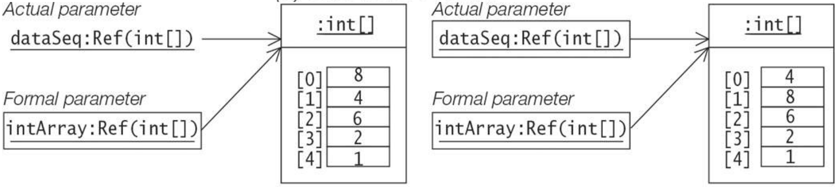

Method call
double totPrice = pizzaHouse.calcPrice(4 , pricePrPizza 15);
Actual parameters
Primitive widening conversion
15.0D
Method definition
public double calcPrice(int numberOfPizzas 4 , double pizzaPrice 15.0D) {
pizzaPrice = pizzaPrice / 2.0;
return numberOfPizzas * pizzaPrice;
}
Formal parameters


Actual parameter
favoritePizza:Ref(Pizza)
:Pizza
meat = "beef"
Copying of reference value creates aliases
pizzaToBeBaked:Ref(Pizza)
Formal parameter

(a) At method call


Actual parameter
favoritePizza:Ref(Pizza)
After return, the actual parameter still denotes the same object whose state has changed
:Pizza
meat = "chicken"
pizzaToBeBaked:Ref(Pizza)
Formal parameter

(b) Just before return


Actual parameter
dataSeq:Ref(int[])
Formal parameter
intArray:Ref(int[])
:int[]
[0] 8
[1] 4
[2] 6
[3] 2
[4] 1
Actual parameter
dataSeq:Ref(int[])
Formal parameter
intArray:Ref(int[])
:int[]
[0] 4
[1] 8
[2] 6
[3] 2
[4] 1

(a) At first call to the `swap()` method (b) Just before first return from the `swap()` method

Declarations 3

Chapter Topics

- An overview of declarations that can be specified in a class
- How to declare and initialize variables
- Using default values for instance variables and static variables
- Understanding lifetime of instance variables, static variables, and local variables
- Using arrays: declaration, construction, initialization, and usage of both simple and multidimensional arrays, including anonymous arrays
- Writing methods, usage of the `this` reference in an instance method, and method overloading
- Understanding the role of constructors, usage of the default constructor, and constructor overloading
- Understanding parameter passing, both primitive values and object references, including arrays and array elements; and declaring `final` parameters0
- Declaring and calling methods with variable arity
- Declaring the `main()` method whose execution starts the application
- Passing program arguments to the `main()` method on the command line
- Declaring and using local variable type inference with `var`

Java SE 17 Developer Exam O	
[3.1] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ○ *Declaring normal classes and defining class members and constructors are covered in this chapter.* ○ *For record classes, see §5.14, p. 299.* ○ *For instance and static initializers, see Chapter 10, p. 531.*	§3.2, p. 99 *to* §3.8, p. 112
[3.2] Implement overloading, including var-arg methods	§3.6, p. 108

§3.11, p. 136	❍ Method overloading and varargs methods are covered in this chapter. ❍ *For comparing overloading and overriding, see §5.1, p. 202.*
§3.13, p. 142	[3.4] Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable ❍ *Local variable type inference is covered in this chapter.* ❍ *For detailed coverage of lambda expressions, see §13.2, p. 679.* ❍ *For variable scope, encapsulation, and immutability, see Chapter 6, p. 323.*
§3.9, p. 117	[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements ❍ *Arrays are covered in this chapter.* ❍ *For Collections and Maps, see Chapter 15, p. 781.*
Java SE 11 Developer Exam O	
§3.13, p. 142	[1.3] Use local variable type inference, including as lambda parameters ❍ *Local variable type inference is covered in this chapter.* ❍ *For detailed coverage of lambda expressions, see §13.2, p. 679.*
§3.2, p. 99 to §3.8, p. 112	[3.2] Define and use fields and methods, including instance, static and overloaded methods
§3.4, p. 102 §3.7, p. 109	[3.3] Initialize objects and their members using instance and static initialiser statements and constructors
§3.9, p. 117	[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods ❍ *Arrays are covered in this chapter.* ❍ *For collections and maps, see Chapter 15, p. 781.*

This chapter covers declaration of classes, methods, constructors, and variables, including array types. Other declarations (e.g., modules, packages, interfaces, enum types, and nested types) will be covered in due course in later chapters.

A *declaration* is a language construct that includes an identifier which can be used to refer to this declaration in the program. Examples include a class declaration that can be referred to by its class name and a method declaration that can be referred to by its method name.

3.1 Class Declarations

A class declaration introduces a new *reference type* and has the following syntax:

Click here to view code image

```
class_modifiers class class_name extends_clause implements_clause // Class header
{ // Class body
    field_declarations
    method_declarations
    constructor_declarations
    member_type_declarations
}
```

In the class header, the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:

- The following *class modifiers*:
  - ○ *Access modifiers*: `public`, `protected`, `private` (§6.5, p. 345)
  - ○ *Non-access class modifiers*: `abstract` (§5.4, p. 218), `final` (§5.5, p. 225), `static` (§9.2, p. 495), `sealed` and `non-sealed` (§5.15, p. 311)
- Any class it *extends* using the `extends` clause (§5.1, p. 191)
- Any interfaces it *implements* using the `implements` clause (§5.6, p. 237)

The class body, enclosed in curly brackets (`{}`), can contain *member declarations*, which comprise the following:

- *Field declarations* (p. 102)
- *Method declarations* (p. 100)
- *Constructor declarations* (p. 109)
- *Member type declarations* (§9.1, p. 491)

The declarations can appear in any order in the class body. The only mandatory parts of the class declaration syntax are the keyword `class`, the class name, and the class body curly brackets (`{}`), as exemplified by the following class declaration:

```
class X { }
```

To understand which code can be legally declared in a class, we distinguish between *static context* and *non-static context*. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, instance field initializers, instance initializer blocks, and constructors. By *static code*, we mean expressions and statements in a static context; by *non-static code*, we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that *static code in a class can only refer to other static members* in the class, whereas *non-static code can refer to any member of the class*.

3.2 Method Declarations

A method declaration has the following syntax:

Click here to view code image

```
method_modifiers return_type method_name
                 (formal_parameter_list)  throws_clause  // Method header
{  // Method body
      local_variable_declarations
      statements
}
```

In addition to the name of the method, the method header can specify the following information:

- The following *method modifiers*:
    - *Access modifiers*: `public`, `protected`, `private` (§6.5, p. 347)
    - *Non-access method modifiers*: `static` (p. 115), `abstract` (§5.4, p. 224), `final` (§5.5, p. 226), `synchronized` (§22.4, p. 1388)
- The *type* of the *return value*, or `void` if the method does not return any value

  A non-`void` method must either use a `return` statement (§4.13, p. 184) to return a value or throw an exception to terminate its execution.
- A *formal parameter list*

  The *formal parameter list* is a comma-separated list of parameters for passing information to the method when the method is invoked by a *method call* (p. 127). An empty parameter list must be specified by `( )`. Each parameter is a simple variable declaration consisting of its type and name:

optional_parameter_modifier type parameter_name

The parameter names are local to the method (§6.6, p. 354). The *optional parameter modifier* `final` is discussed in §3.10, p. 135. It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a method.

The *type* in the parameter declaration *cannot* be designated by the `var` reserved type name, as illustrated in the method declaration below. At compile time, it is not possible to determine the type of the `newSpeed` formal parameter.

Click here to view code image

```
void setSpeed(var newSpeed) {} // var not permitted. Compile-
time error!
```

- Any *exceptions* thrown by the method, which are specified in a `throws` clause (§7.5, p. 388)

The method body is a *block* (`{}`) containing the *local variable declarations* (p. 102) and the *statements* (p. 101) of the method. The `return` statement in the body of a method is of particular importance as it terminates the execution of the method and can optionally return a value to the caller of the method (§4.13, p. 184).

The mandatory parts of a method declaration are the return type, the method name, and the method body curly brackets (`{}`), as exemplified by the following method declaration:

```
void noAction() {}
```

Member methods are characterized as one of two types: *instance methods* (p. 106) and *static methods* (p. 112).

The *signature* of a method comprises the name of the method and the types of the formal parameters only. The following method:

Click here to view code image

```
double cubeVolume(double length, double width, double height) {}
```

has the signature:

Click here to view code image

```
cubeVolume(double, double, double)
```

3.3 Statements

Statements in Java can be grouped into various categories. Variable declarations, optionally specified with an initialization expression, are called *declaration statements* (p. 102). Other basic forms of statements are *control flow statements* (Chapter 4, p. 151) and *expression statements*.

An *expression statement* is an expression terminated by a semicolon (`;`). Any value returned by the expression is discarded. Only certain types of expressions have meaning as statements:

- Assignments (§2.7, p. 54)
- Increment and decrement operators (§2.10, p. 69)
- Method calls (p. 127)
- Object creation expressions with the `new` operator (§2.19, p. 92)

A solitary semicolon denotes the *empty statement*, which does nothing.

A block, `{}`, is a *compound* statement that can be used to group zero or more local declarations and statements (§6.6, p. 354). Blocks can be nested, since a block is a statement that can contain other statements. A block can be used in any context where a simple statement is permitted. The compound statement that is embodied in a block begins at the left curly bracket (`{`) and ends with a matching right curly bracket (`}`). Such a block must not be confused with an array initializer in declaration statements (p. 119).

Labeled statements are discussed in §4.10, p. 179.

3.4 Variable Declarations

A *variable* stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can store only values of primitive data types and reference values of objects. Variables that store reference values of objects are called *reference variables* (or *object references* or simply *references*).

We distinguish between two kinds of variables: *field variables* and *local variables*. Field variables are variables that are declared in *type declarations* (*classes*, *interfaces*, and *enums*). Local variables are variables that are declared in *methods*, *constructors*, and *blocks*. Local variables declared with the `var` type name are discussed in §3.13, p. 142.

Declaring and Initializing Variables

Variable declarations (technically called *declaration statements*) are used to *declare* variables, meaning they are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. Examples of declaring variables that can store primitive values follow:

Click here to view code image

```
char a, b, c;            // a, b, and c are character variables.
double area;             // area is a floating-point variable.
boolean flag;            // flag is a boolean variable.
```

The first declaration is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also be combined with an *initialization expression* to specify an appropriate initial value for the variable.

Click here to view code image

```
int i = 10,              // i is an int variable with initial value 10.
    j = 0b101;           // j is an int variable with initial value 5.
long big = 2147483648L;   // big is a long variable with specified initial val
```

Reference Variables

A *reference variable* can store the reference value of an object, and can be used to manipulate the object denoted by the reference value.

A variable declaration that specifies a *reference type* (i.e., a class, an array, an interface name, or an enum type) declares a reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration specifies the name and the reference type only. The declaration determines which objects can be referenced by a reference variable. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

Click here to view code image

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizz
Hamburger bigOne,   // Variable bigOne can reference objects of class Hamburge
          smallOne; // and so can variable smallOne.
```

It is important to note that the preceding declarations do not create any objects of class `Pizza` or `Hamburger`. Rather, they simply create variables that can store reference values of objects of the specified classes.

A declaration can also be combined with an *initializer expression* to create an object whose reference value can be assigned to the reference variable:

Click here to view code image

```
Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration statement
```

The reference variable `yummyPizza` can reference objects of class `Pizza`. The keyword `new`, together with the *constructor call* `Pizza("Hot&Spicy")`, creates an object of the class `Pizza`. The reference value of this object is assigned to the variable `yummyPizza`. The newly created object of class `Pizza` can now be manipulated through the reference variable `yummyPizza`.

Initial Values for Variables

This section discusses what value, if any, is assigned to a variable when no explicit initial value is provided in the declaration.

Default Values for Fields

Default values for fields of primitive data types and reference types are listed in Table 3.1. The value assigned depends on the type of the field.

Table 3.1 *Default Values*

Data type	**Default value**
`boolean`	`false`
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0L` for `long`, `0` for others
Floating-point (`float`, `double`)	`0.0F` or `0.0D`
Reference types	`null`

If no explicit initialization is provided for a static variable, it is initialized with the default value of its type when the class is loaded. Similarly, if no initialization is provided for an instance variable, it is initialized with the default value of its type when the class is instantiated. The fields of reference types are always initialized with the null reference value if no initialization is provided.

Example 3.1 illustrates the default initialization of fields. Note that static variables are initialized when the class is loaded the first time, and instance variables are initialized accordingly in *every* object created from the class `Light`.

Example 3.1 *Default Values for Fields*

Click here to view code image

```
public class Light {
  // Static variable
  static int counter;      // Default value 0 when class is loaded

  // Instance variables:
  int     noOfWatts = 100; // Explicitly set to 100
  boolean indicator;       // Implicitly set to default value false
  String  location;        // Implicitly set to default value null

  public static void main(String[] args) {
    Light bulb = new Light();
    System.out.println("Static variable counter:     " + Light.counter);
    System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
    System.out.println("Instance variable indicator: " + bulb.indicator);
    System.out.println("Instance variable location:  " + bulb.location);
  }
}
```

Output from the program:

Click here to view code image

```
Static variable counter:     0
Instance variable noOfWatts: 100
Instance variable indicator: false
Instance variable location:  null
```

Initializing Local Variables of Primitive Data Types

Local variables are variables that are declared in *methods*, *constructors*, and *blocks*. They are *not* initialized implicitly when they are allocated memory at method invocation—that is, when the execution of a method begins. The same applies to local variables in constructors and blocks. Local variables must be explicitly initialized before being used. The compiler will report an error only if an attempt is made to *use* an uninitialized local variable.

Example 3.2 *Flagging Uninitialized Local Variables of Primitive Data Types*

Click here to view code image

```
public class TooSmartClass {
  public static void main(String[] args) {
    double weight = 10.0, thePrice;                    // (1) Local variables

    if (weight <  10.0) thePrice = 20.50;
    if (weight >  50.0) thePrice = 399.00;
    if (weight >= 10.0) thePrice = weight * 10.0;      // (2) Always executed
    System.out.println("The price is: " + thePrice);   // (3) Compile-time err
  }
}
```

In Example 3.2, the compiler complains that the local variable `thePrice` used in the print statement at (3) may not have been initialized. If allowed to compile and execute, the local variable `thePrice` will get the value `100.0` in the last `if` statement at (2), before it is used in the print statement. However, in Example 3.2, the compiler cannot guarantee that code in the body of any of the `if` statements will be executed and thereby the local variable `thePrice` is initialized, as it cannot determine whether the condition in any of the `if` statements is `true` at compile time.

We will not go into the details of *definite assignment analysis* that the compiler performs to guarantee that a local variable is initialized before it is used. In essence, the compiler determines whether a variable is initialized on a path of control flow from where it is declared to where it is used. This analysis can at times be conservative, as in Example 3.2.

The program will compile correctly if the local variable `thePrice` is initialized in the declaration, or if an unconditional assignment is made to it. Replacing the declaration of the local variables at (1) in Example 3.2 with the following declaration solves the problem:

Click here to view code image

```
double weight = 10.0, thePrice = 0.0; //    (1') Both local variables initiali
```

Replacing the condition in any of the `if` statements with a *constant expression* that evaluates to `true` will also allow the compiler to ensure that the local variable `thePrice` is initialized before use in the print statement.

Initializing Local Reference Variables

Local reference variables are bound by the same initialization rules as local variables of primitive data types.

Example 3.3 *Flagging Uninitialized Local Reference Variables*

Click here to view code image

```
public class VerySmartClass {
  public static void main(String[] args) {
    String importantMessage;       // Local reference variable

    System.out.println("The message length is: " +
                       importantMessage.length());  // Compile-time error!
  }
}
```

In Example 3.3, the compiler complains that the local variable `importantMessage` used in the `println` statement may not be initialized. If the variable `important-Message` is set to the value `null`, the program will compile. However, a runtime error (`NullPointerException`) will occur when the code is executed because the variable `importantMessage` will not denote any object. The golden rule is to ensure that a reference variable, whether local or not, is assigned a reference value denoting an object before it is used—that is, to ensure that it does not have the value `null`.

The program compiles and runs if we replace the declaration with the following declaration of the local variable, which creates a string literal and assigns its reference value to the local reference variable `importantMessage`:

Click here to view code image

```
String importantMessage = "Initialize before use!";
```

Arrays and their default values are discussed in §3.9, p. 119.

Lifetime of Variables

The *lifetime* of a variable—that is, the time a variable is accessible during execution—is determined by the context in which it is declared. The lifetime of a variable, which is also called its *scope*, is discussed in more detail in §6.6, p. 352. We distinguish among the lifetime of variables in the following three contexts:

- *Instance variables*: members of a class, which are created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the *state* of the object. Instance variables exist as long as the object they belong to is in use at runtime.

- *Static variables*: members of a class, but which are not created for any specific object of the class, and therefore, belong only to the class. They are created when the class is loaded at runtime and exist as long as the class is available at runtime.
- *Local variables* (also called *method automatic variables*): declared in methods, constructors, and blocks, and created for each execution of the method, constructor, or block. After the execution of the method, constructor, or block completes, local variables are no longer accessible.

3.5 Instance Methods and the Object Reference `this`

Instance methods belong to every object of the class and can be invoked only on objects. All members defined in the class, both static and non-static, are accessible in the context of an instance method. The reason is that all instance methods are passed an implicit reference to the *current object*—that is, the object on which the method is being invoked.

The current object can be referenced in the body of the instance method by the keyword `this`. In the body of the method, the `this` reference can be used like any other object reference to access members of the object. In fact, the keyword `this` can be used in any non-static context. The `this` reference can be used as a normal reference to reference the current object, but the reference cannot be modified—it is a `final` reference (§5.5, p. 225).

The `this` reference to the current object is useful in situations where a local variable hides, or *shadows*, a field with the same name. In Example 3.4, the two parameters `noOfWatts` and `indicator` in the constructor of the `Light` class have the same names as the fields in the class. The example also declares a local variable `location`, which has the same name as one of the fields. The reference `this` can be used to distinguish the fields from the local variables. At (1), the `this` reference is used to identify the field `noOfWatts`, which is assigned the value of the parameter `noOfWatts`. Without the `this` reference at (2), the value of the parameter `indicator` is assigned back to this parameter, and not to the field by the same name, resulting in a logical error. Similarly at (3), without the `this` reference, it is the local variable `location` that is assigned the value of the parameter `site`, and not the field with the same name.

Example 3.4 *Using the* `this` *Reference*

Click here to view code image

```
public class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // Constructor:
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;

    this.noOfWatts = noOfWatts;    // (1) Assignment to field
    indicator = indicator;         // (2) Assignment to parameter
    location = site;               // (3) Assignment to local variable
    this.superfluous();            // (4)
    superfluous();                 // equivalent to call at (4)
  }

  // Instance method:
  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  // Static method:
  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

Probable output from the program:

Click here to view code image

```
Current object: Light@1bc4459
Current object: Light@1bc4459
No. of watts: 100
Indicator:    false
Location:     null
```

If a member is not shadowed by a local declaration, the simple name `member` is considered a shorthand notation for `this.member`. In particular, the `this` reference can be used explicitly to invoke other methods in the class. This usage is illustrated at (4) in Example 3.4, where the method `superfluous()` is called.

If, for some reason, a method needs to pass the current object to another method, it can do so using the `this` reference. This approach is illustrated at (5) in Example 3.4, where the current object is passed to the `printf()` method. The `printf()` method prints the text representation of the current object (which comprises the name of the class of the current object and the hexadecimal representation of the current object's hash code). The *hash code* of an object is an `int` value that uniquely identifies the object.

Note that the `this` reference cannot be used in a static context, as static code is not executed in the context of any object.

3.6 Method Overloading

Each method has a *signature*, which comprises the name of the method plus the types and order of the parameters in the formal parameter list. Several method implementations may have the same name, as long as the method signatures differ. This practice is called *method overloading*. Because overloaded methods have the same name, their parameter lists must be different.

Rather than inventing new method names, method overloading can be used when the same logical operation requires multiple implementations. The Java SE Platform API makes heavy use of method overloading. For example, the class `java.lang.Math` contains an overloaded method `min()`, which returns the minimum of two numeric values.

Click here to view code image

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

In the following examples, five implementations of the method `methodA` are shown:

Click here to view code image

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

The corresponding signatures of the five methods are as follows:

Click here to view code image

```
methodA(int, double)       1'
methodA(int)j              2': Number of parameters
methodA()                  3': Number of parameters
methodA(double, int)       4': Order of parameters
methodA(int, double)       5': Same as 1'
```

The first four implementations of the method named `methodA` are overloaded correctly, each time with a different parameter list and, therefore, different signatures. The declaration at (5) has the same signature `methodA(int, double)` as the declaration at (1), and therefore, is not a valid overloading of this method.

Click here to view code image

```
void bake(Cake k)  { /* ... */ }                  // (1)
void bake(Pizza p) { /* ... */ }                  // (2)

int     halfIt(int a) { return a/2; }             // (3)
double  halfIt(int a) { return a/2.0; }           // (4) Not OK. Same signature
```

The method named `bake` is correctly overloaded at (1) and (2), with two different parameter lists. In the implementation, changing just the return type (as shown at (3) and (4) in the preceding example) is not enough to overload a method and will be flagged as a compile-time error. The parameter list in the declarations must be different.

Only methods declared in the same class and those that are inherited by the class can be overloaded. Overloaded methods should be considered to be individual methods that just happen to have the same name. Methods with the same name are allowed, since methods are identified by their signature. At compile time, the right implementation of an overloaded method is chosen, based on the signature of the method call. Details of method overloading resolution can be found in §5.10, p. 265. Method overloading should not be confused with *method overriding* (§5.1, p. 196).

3.7 Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the `new` operator.

The following simplified syntax is the canonical declaration of a constructor:

Click here to view code image

```
access_modifier class_name  (formal_parameter_list)
                          throws_clause   // Constructor header
```

```
  { // Constructor body
        local_variable_declarations
        statements
  }
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an access modifier are not permitted in the constructor header. For more on access modifiers for constructors, see §6.5, p. 345.
- Constructors cannot return a value, and therefore, do not specify a return type, not even `void`, in the constructor header. But their declaration can use the `return` statement (without the return value) in the constructor body (§4.13, p. 184).
- The constructor name must be the same as the class name.

Class names and method names exist in different *namespaces*. Thus there are no name conflicts in Example 3.5, where a method declared at (2) has the same name as the constructor declared at (1). The method `Name()` at (2) is also breaking the convention of starting a method name with a lowercase character. A method must always specify a return type, whereas a constructor does not. However, using such naming schemes is strongly discouraged.

A constructor that has no parameters, like the one at (1) in Example 3.5, is called a *no-argument constructor*.

Example 3.5 *Namespaces*

Click here to view code image

```
public class Name {

  Name() {                          // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                     // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();              // (3) Constructor call followed by method cal
  }
}
```

Output from the program:

```
Constructor
Method
```

The Default Constructor

If a class does not specify *any* constructors, then a *default constructor* is generated for the class by the compiler. The default constructor is equivalent to the following implementation:

Click here to view code image

```
class_name() { super(); }   // No parameters. Calls superclass constructor.
```

A default constructor is a no-argument constructor. The only action taken by the default constructor is to call the superclass constructor. This ensures that the inherited state of the object is initialized properly (§5.3, p. 209). In addition, all instance variables in the object are set to the default value of their type, barring those that are initialized by an initialization expression in their declaration.

In the following code, the class `Light` does not specify any constructors:

Click here to view code image

```
class Light {
  // Fields:
  int     noOfWatts;        // Wattage
  boolean indicator;        // On or off
  String  location;         // Placement

  // No constructors
  //...
}

class Greenhouse {
  // ...
  Light oneLight = new Light();     // (1) Call to default constructor
}
```

In this code, the following default constructor is called when a `Light` object is created by the object creation expression at (1):

```
Light() { super(); }
```

Creating an object using the `new` operator with the default constructor, as at (1), will initialize the fields of the object to their *default values* (i.e., the fields `noOfWatts`, `indicator`, and `location` in a `Light` object will be initialized to `0`, `false`, and `null`, respectively).

A class can choose to provide its own constructors, rather than relying on the default constructor. In the following example, the class `Light` provides a no-argument constructor at (1).

Click here to view code image

```
class Light {
  // ...
  Light() {                        // (1) No-argument constructor
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  //...
}
class Greenhouse {
  // ...
  Light extraLight = new Light();   // (2) Call to no-argument constructor
}
```

The no-argument constructor ensures that any object created with the object creation expression `new Light()`, as at (2), will have its fields `noOfWatts`, `indicator`, and `location` initialized to `50`, `true`, and `"X"`, respectively.

If a class defines *any* constructor, the default constructor is not generated. If such a class requires a no-argument constructor, it must provide its own implementation, as in the preceding example. In the next example, the class `Light` does not provide a no-argument constructor, but rather includes a non-zero argument constructor at (1). It is called at (2) when an object of the class `Light` is created with the `new` operator. Any attempt to call the default constructor will be flagged as a compile-time error, as shown at (3).

Click here to view code image

```
class Light {
  // ...
  // Only non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {          // (1)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");// (2) OK
  Light firstLight = new Light();                       // (3) Compile-time er
}
```

Overloaded Constructors

Like methods, constructors can be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists. In the following example, the class `Light` now provides explicit implementation of the no-argument constructor at (1) and that of a non-zero argument constructor at (2). The constructors are overloaded, as is evident by their signatures. The non-zero argument constructor at (2) is called when an object of the class `Light` is created at (4), and the no-argument constructor is likewise called at (3). Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked (see chaining of constructors in §5.3, p. 209). It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a constructor.

Click here to view code image

```
class Light {
  // ...
  // No-argument constructor:
  Light() {                                                  // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light firstLight = new Light();                        // (3) OK. Calls (1)
  Light moreLight  = new Light(100, true, "Greenhouse"); // (4) OK. Calls (2)
}
```

3.8 Static Member Declarations

In this section we look at static members in classes, but in general, the keyword `static` is used in the following contexts:

- Declaring `static` fields in classes (p. 113), enum types (§5.13, p. 290) and interfaces (§5.6, p. 254)

- Declaring `static` methods in classes (p. 115), enum types (§5.13, p. 290) and interfaces (§5.6, p. 251)
- Declaring `static` initializer blocks in classes (§10.7, p. 545) and enum types (§5.13, p. 290)
- Declaring nested `static` member types (§9.2, p. 495)

Static Members in Classes

Static members belong to the class in which they are declared and are not part of any instance of the class. The declaration of `static` members is prefixed by the keyword `static` to distinguish them from instance members.

Static code inside a class can access a `static` member in the following three ways:

- By the `static` member's simple name
- By using the class name with the `static` member's name
- By using an object reference of the `static` member's class with the `static` member's name

Depending on the access modifier of the `static` members declared in a class, clients can only access these members by using the class name or using an object reference of their class.

The class need not be instantiated to access its `static` members. This is in contrast to instance members of the class which can only be accessed by references that actually refer to an instance of the class.

Static Fields in Classes

Static fields (also called *static variables* and *class variables*) exist only in the class in which they are defined. When the class is loaded, `static` fields are initialized to their default values if no explicit initialization is specified. They are not created when an instance of the class is created. In other words, the values of these fields are not a part of the state of any object. Static fields are akin to global variables that can be shared with all objects of the class and with other clients, if necessary.

Example 3.6 *Accessing Static Members in a Class*

Click here to view code image

```
// File: StaticTest.java
import static java.lang.System.out;

class Light {

  // Static field:
  static int counter;                  // (1) No initializer expression

  // Static method:
  public static void printStatic() {
    Light myLight = null;
    out.printf("%s, %s, %s%n", counter, Light.counter, myLight.counter); // (2
    long counter = 10;                 // (3) Local variable shadows static fi
    out.println("Local counter: " + counter);       // (4) Local variable acce
    out.println("Static counter: " + Light.counter);// (5) Static field access

//  out.println(this.counter);         // (6) Cannot use this in static contex
//  printNonStatic();                  // (7) Cannot call non-static method
  }

  // Non-static method:
  public void printNonStatic() {
   out.printf("%s, %s, %s%n", counter, this.counter, Light.counter);     // (8
  }
}
//________________________________________________________________________________
public class StaticTest {              // Client of class Light
  public static void main(String[] args) {
    Light.counter++;                   // (9) Using class name
    Light dimLight = null;
    dimLight.counter++;                // (10) Using object reference

    out.print("Light.counter == dimLight.counter: ");
    out.println(Light.counter == dimLight.counter);//(11) Aliases for static f

    out.println("Calling static method using class name:");
    Light.printStatic();               // (12) Using class name
    out.println("Calling static method using object reference:");
    dimLight.printStatic();            // (13) Using object reference
  }
}
```

Output from the program:

Click here to view code image

```
Light.counter == dimLight.counter: true
Calling static method using class name:
2, 2, 2
Local counter: 10
Static counter: 2
Calling static method using object reference:
2, 2, 2
Local counter: 10
Static counter: 2
```

In Example 3.6, the `static` field `counter` at (1) will be initialized to the default value `0` when the class is loaded at runtime, since no initializer expression is specified. The print statement at (2) in the `static` method `printCount()` shows how this `static` field can be accessed in three different ways, respectively: simple name `counter`, the class name `Light`, and object reference `myLight` of class `Light`, although no object has been created.

Shadowing of fields by local variables is different from *hiding* of fields by field declarations in subclasses. In Example 3.6, a local variable is declared at (3) that has the same name as the `static` field. Since this local variable *shadows* the `static` field, the simple name at (4) now refers to the local variable, as shown by the output from the program. The shadowed `static` field can of course be accessed using the class name, as shown at (5). It is the local variable that is accessed by its simple name as long as it is in scope.

Trying to access the `static` field with the `this` reference at (6) results in a compile-time error, since the `this` reference cannot be used in static code. Invoking the non-`static` method at (7) also results in a compile-time error, since static code cannot refer to non-`static` members by its simple name in the class.

The print statement at (8) in the method `printNonStatic()` illustrates referring to `static` members in non-static code: It refers to the `static` field counter by its simple name, with the `this` reference, and using the class name.

In Example 3.6, the class `StaticTest` is a client of the class `Light`. The client must use the class name or an object reference of class `Light` at (9) and (10), respectively, to access the `static` field `counter` in the class `Light`. The result from the print statement at (11) shows that these two ways of accessing a `static` field are equivalent.

Static Methods in Classes

Static methods are also known as *class methods*. A `static` method in a class can directly access other `static` members in the class by their simple name. It cannot access instance (i.e., non-`static`) members of the class directly, as there is no notion of an object associated with a `static` method.

A typical `static` method might perform some task on behalf of the whole class or for objects of the class. Static methods are often used to implement *utility classes* that provide common and frequently used functions. A good example of a utility class is the `java.lang.Math` class in the Java platform SE API that provides common mathematical functions.

Static methods can be overloaded analogous to instance methods. Static methods in a superclass cannot be overridden in a subclass as instance methods can, but they can be *hidden* by `static` methods in a subclass (§5.1, p. 203).

A type parameter of a generic class or interface cannot be used in a `static` method (§11.2, p. 567).

Example 3.6 shows how the `static` method `printCount()` of the class `Light` can be invoked using the class name and via an object reference of the class `Light` at (12) and (13), respectively.

Review Questions

3.1 In which of these variable declarations will the variable remain uninitialized unless it is explicitly initialized?

Select the one correct answer.

a. Declaration of an instance variable of type `int`
b. Declaration of a static variable of type `float`
c. Declaration of a local variable of type `float`
d. Declaration of a static variable of type `Object`
e. Declaration of an instance variable of type `int[]`

3.2 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Init {

  String title;
  boolean published;

  static int total;
  static double maxPrice;

  public static void main(String[] args) {
    Init initMe = new Init();
    double price;
    if (true)
      price = 100.00;
    System.out.println("|" + initMe.title + "|" + initMe.published + "|"
                             Init.total + "|" + Init.maxPrice + "|" + price +
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will print `|null|false|0|0.0|0.0|` at runtime.

c. The program will print `|null|true|0|0.0|100.0|` at runtime.

d. The program will print `| |false|0|0.0|0.0|` at runtime.

e. The program will print `|null|false|0|0.0|100.0|` at runtime.

3.3 Given that the following pairs of method are declared in the same class, which of the following statements are true?

Click here to view code image

```
void fly(int distance) {}
int  fly(int time, int speed) { return time*speed; }

void fall(int time) {}
int  fall(int distance) { return distance; }

void glide(int time) {}
void Glide(int time) {}
```

Select the two correct answers.

a. The first pair of methods will compile and will overload the method name `fly`.

b. The second pair of methods will compile and will overload the method name `fall`.

c. The third pair of methods will compile and will overload the method name `glide`.

d. The first pair of methods will fail to compile.

e. The second pair of methods will fail to compile.

f. The third pair of methods will fail to compile.

3.4 Which of the following statements are true? Select the two correct answers.

a. A class must define a constructor.

b. A constructor can be declared `private`.

c. A constructor can return a value.

d. A constructor must initialize all fields when a class is instantiated.

e. A constructor can access the non-static members of a class.

3.5 What will be the result of compiling the following program?

Click here to view code image

```
public class MyClass {
  long var;

  public void MyClass(long param) { var = param; }  // (1)

  public static void main(String[] args) {
    MyClass a, b;
    a = new MyClass();                               // (2)
    b = new MyClass(5);                              // (3)
  }
}
```

Select the one correct answer.

a. A compile-time error will occur at (1).
b. A compile-time error will occur at (2).
c. A compile-time error will occur at (3).
d. The program will compile without errors.

3.6 Which statement is true?

Select the one correct answer.

a. A `static` method can call other non-`static` methods in the same class by using the `this` keyword.
b. A class may contain both `static` and non-`static` variables, and both `static` and non-`static` methods.
c. Each object of a class has its own instance of the `static` variables declared in the class.
d. Instance methods may access local variables of `static` methods.
e. All methods in a class are implicitly passed the `this` reference as an argument, when invoked.

3.9 Arrays

An *array* is a data structure that defines an indexed collection with a fixed number of data elements that all have the *same type*. A position in the array is indicated by a non-negative integer value called the *index*. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot be changed after the array has been created.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a `public final` field called `length`, which specifies the array size (i.e., the number of elements the array can accommodate). The first element is always at index 0 and the last element at index *n* – 1, where *n* is the value of the `length` field in the array.

Simple arrays are *one-dimensional arrays*—that is, a simple list of values. Since arrays can store reference values, the objects referenced can also be array objects. Thus a multidimensional arrays is implemented as an *array of arrays* (p. 124).

Passing array references as parameters is discussed in §3.10, p. 127. Type conversions for array references on assignment and on method invocation are discussed in §5.9, p. 261, and §5.10, p. 265, respectively.

Declaring Array Variables

A one-dimensional array variable declaration has either of the following syntaxes:

element_type `[]` *array_name* `;`

or

element_type array_name `[];`

where *element_type* can be a primitive data type or a reference type. The array variable *array_name* has the type *element_type*`[]`. Note that the array size is not specified. As a consequence, the array variable *array_name* can be assigned the reference value of an array of any length, as long as its elements have *element_type*.

It is important to understand that the declaration does not actually create an array. Instead, it simply declares a *reference* that can refer to an array object. The `[]` notation can also be specified after a variable name to declare it as an array variable, but then it applies to just that variable.

Click here to view code image

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

These two declarations declare `anIntArray` and `mediumPizzas` to be reference variables that can refer to arrays of `int` values and arrays of `Pizza` objects, respectively. The variable `largePizzas` can denote an array of `Pizza` objects, but the variable `oneInteger` cannot denote an array of `int` values—it is a simple variable of the type `int`.

An array variable that is declared as a field in a class, but is not explicitly initialized to any array, will be initialized to the default reference value `null`. This default initialization does *not* apply to *local* reference variables, and therefore, does not apply to local array variables either. This behavior should not be confused with initialization of the elements of an array during array construction.

Constructing an Array

An array can be constructed for a fixed number of elements of a specific type, using the `new` operator. The reference value of the resulting array can be assigned to an array variable of the corresponding type. The syntax of the *array creation expression* is shown on the right-hand side of the following assignment statement:

Click here to view code image

```
array_name = new element_type[array_size];
```

The minimum value of *array_size* is `0`; in other words, zero-length arrays can be constructed in Java. If the array size is negative, a `NegativeArraySizeException` is thrown at runtime.

Given the declarations

Click here to view code image

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

the three arrays in the declarations can be constructed as follows:

Click here to view code image

```
anIntArray   = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
largePizzas  = new Pizza[3];         // array of 3 pizzas
```

The array declaration and construction can be combined.

Click here to view code image

```
element_type1[] array_name = new element_type2[array_size];
```

In the preceding syntax, the array type *element_type*$_2$`[]` must be *assignable* to the array type *element_type*$_1$`[]` (§5.8, p. 261). When the array is constructed, all of its elements are initialized to the default value for $\mathit{element\_type}_2$. This is true for both member and local arrays when they are constructed.

In the following examples, the code constructs the array, and the array elements are implicitly initialized to their default values. For example, all elements of the array `anIntArray` get the value `0`, and all elements of the array `mediumPizzas` get the value `null` when the arrays are constructed.

Click here to view code image

```
int[] anIntArray = new int[10];                  // Default element value: 0
Pizza[] mediumPizzas = new Pizza[5];             // Default element value: nul
```

The value of the field `length` in each array is set to the number of elements specified during the construction of the array; for example, `mediumPizzas.length` has the value `5`.

Once an array has been constructed, its elements can also be explicitly initialized individually—for example, in a loop. The examples in the rest of this section make use of a loop to iterate over the elements of an array for various purposes.

Initializing an Array

Java provides the means to declare, construct, and explicitly initialize an array in one declaration statement:

Click here to view code image

```
element_type[] array_name = { array_initialize_list };
```

This form of initialization applies to fields as well as to local arrays. The *array_initialize_list* is a comma-separated list of zero or more expressions. Such an array initializer results in the construction and initialization of the array.

Click here to view code image

```
int[] anIntArray = {13, 49, 267, 15, 215};
```

In the declaration statement above, the variable `anIntArray` is declared as a reference to an array of `int`s. The array initializer results in the construction of an array to hold five elements (equal to the length of the list of expressions in the block), where the first element is initialized to the value of the first expression (`13`), the second element to the value of the second expression (`49`), and so on.

Click here to view code image

```
Pizza[] pizzaOrder = { new Pizza(), new Pizza(), null };
```

In this declaration statement, the variable `pizzaOrder` is declared as a reference to an array of `Pizza` objects. The array initializer constructs an array to hold three elements. The initialization code sets the first two elements of the array to refer to two `Pizza` objects, while the last element is initialized to the null reference. The reference value of the array of `Pizza` objects is assigned to the reference `pizzaOrder`. Note also that this declaration statement actually creates *three* objects: the array object with three references and the two `Pizza` objects.

The expressions in the *array_initialize_list* are evaluated from left to right, and the array name obviously cannot occur in any of the expressions in the list. In the preceding examples, the *array_initialize_list* is terminated by the right curly bracket, `}`, of the block. The list can also be legally terminated by a comma. The following array has length 2, and not 3:

Click here to view code image

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

The declaration statement at (1) in the following code defines an array of four `String` objects, while the declaration statement at (2) shows that a `String` object is not the same as an array of `char`.

Click here to view code image

```
// Array with 4 String objects:
String[] pets = {"crocodiles", "elephants", "crocophants", "elediles"}; // (1)

// Array of 3 characters:
char[] charArray = {'a', 'h', 'a'};    // (2) Not the same as "aha"
```

Using an Array

The array object is referenced by the array name, but individual array elements are accessed by specifying an index with the `[]` operator. The array element access expression has the following syntax:

Click here to view code image

```
array_name [index_expression]
```

Each individual element is treated as a simple variable of the element type. The *index* is specified by the *index_expression*, whose value should be promotable to an `int` value; otherwise, a compile-time error is flagged. Since the lower bound of an array index is always `0`, the upper bound is 1 less than the array size—that is, *array_name*`.length-1`. The `i`th element in the array has index `(i-1)`. At runtime, the index value is automatically checked to ensure that it is within the array index bounds. If the index value is less than `0`, or greater than or equal to *array_name*`.length`, an `ArrayIndexOutOfBoundsException` is thrown. A program can either explicitly check that the index value is within the array index bounds or catch the runtime exception that is thrown if it is invalid (§7.3, p. 375), but an illegal index is typically an indication of a programming error.

In the array element access expression, the *array_name* can be any expression that returns a reference to an array. For example, the expression on the right-hand side of the following assignment statement returns the character `'H'` at index 1 in the character array returned by a call to the `toCharArray()` method of the `String` class:

Click here to view code image

```
char letter = "AHA".toCharArray()[1];     // 'H'
```

The array operator `[]` is used to declare array types (`Topping[]`), specify the array size (`new Topping[3]`), and access array elements (`toppings[1]`). This operator is not used when the array reference is manipulated, such as in an array reference assignment, or when the array reference is passed as an actual parameter in a method call (p. 132).

Example 3.7 shows traversal of arrays using `for` loops (§4.7, p. 174 and p. 176). A `for(;;)` loop at (3) in the `main()` method initializes the local array `trialArray` declared at (2) five times with pseudorandom numbers (from `0.0` to `100.0`), by calling the method `randomize()` declared at (5). The minimum value in the array is found by calling the method `findMinimum()` declared at (6), and is stored in the array `storeMinimum` declared at (1). Both of these methods also use a `for(;;)` loop. The loop variable is initialized to a start value—`0` at (3) and (5), and `1` at (6). The loop condition tests whether the loop variable is less than the length of the array; this guarantees that the loop will terminate when the last element has been accessed. The loop variable is incremented after each iteration to access the next element.

A `for(:)` loop at (4) in the `main()` method is used to print the minimum values from the trials, as elements are read consecutively from the array, without keeping track of an index value.

Example 3.7 *Using Arrays*

Click here to view code image

```
public class Trials {
  public static void main(String[] args) {
    // Declare and construct the local arrays:
    double[] storeMinimum = new double[5];               // (1)
    double[] trialArray = new double[15];                // (2)
    for (int i = 0; i < storeMinimum.length; ++i) {      // (3)
      // Initialize the array.
      randomize(trialArray);

      // Find and store the minimum value.
      storeMinimum[i] = findMinimum(trialArray);
    }

    // Print the minimum values:                            (4)
    for (double minValue : storeMinimum)
      System.out.printf("%.4f%n", minValue);
  }

  public static void randomize(double[] valArray) {      // (5)
    for (int i = 0; i < valArray.length; ++i)
      valArray[i] = Math.random() * 100.0;
  }

  public static double findMinimum(double[] valArray) {  // (6)
    // Assume the array has at least one element.
    double minValue = valArray[0];
    for (int i = 1; i < valArray.length; ++i)
      minValue = Math.min(minValue, valArray[i]);
    return minValue;
  }
}
```

Probable output from the program:

```
6.9330
2.7819
6.7427
18.0849
26.2462
```

Anonymous Arrays

As shown earlier in this section, the following declaration statement can be used to construct arrays using an array creation expression:

Click here to view code image

```
element_type₁[] array_name = new element_type₂[array_size];    // (1)
```

```
int[] intArray = new int[5];
```

The size of the array is specified in the array creation expression, which creates the array and initializes the array elements to their default values. By comparison, the following declaration statement both creates the array and initializes the array elements to specific values given in the array initializer:

Click here to view code image

```
element_type[] array_name = { array_initialize_list }; // (2)
```

```
int[] intArray = {3, 5, 2, 8, 6};
```

However, the array initializer is *not* an expression. Java has another array creation expression, called an *anonymous array*, which allows the concept of the array creation expression from (1) to be combined with the array initializer from (2), so as to create and initialize an array:

Click here to view code image

```
new element_type[] { array_initialize_list }
```

```
new int[] {3, 5, 2, 8, 6}
```

This construct has enough information to create a nameless array of a specific type and specific length. Neither the name of the array nor the size of the array is specified. The construct returns the reference value of the newly created array, which can be assigned to references and passed as arguments in method calls. In particular, the following declaration statements are equivalent:

Click here to view code image

```
int[] intArray = {3, 5, 2, 8, 6};                          // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6};                // (2)
```

At (1), an array initializer is used to create and initialize the elements. At (2), an anonymous array expression is used. It is tempting to use the array initializer as an expression—for example, in an assignment statement, as a shortcut for assigning values to array elements in one go. However, this is not allowed; instead, an anonymous array expression should be used. The concept of the anonymous array combines the definition and the creation of the array into one operation.

Click here to view code image

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31};                   // Compile-time erro
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // O
```

In Example 3.8, an anonymous array is constructed at (1), and passed as an actual parameter to the static method `findMinimum()` defined at (2). Note that no array name or array size is specified for the anonymous array.

Example 3.8 *Using Anonymous Arrays*

Click here to view code image

```
public class AnonArray {
  public static void main(String[] args) {
    System.out.println("Minimum value: " +
        findMinimum(new int[] {3, 5, 2, 8, 6}));                   // (1)
  }

  public static int findMinimum(int[] dataSeq) {                   // (2)
    // Assume the array has at least one element.
    int min = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index] < min)
        min = dataSeq[index];
    return min;
  }
}
```

Output from the program:

```
Minimum value: 2
```

Multidimensional Arrays

Since an array element can be an object reference and arrays are objects, array elements can themselves refer to other arrays. In Java, an array of arrays can be defined as follows:

Click here to view code image

element_type `[][]...[]` *array_name*`;`

or

Click here to view code image

element_type array_name `[][]...[];`

In fact, the sequence of square bracket pairs, `[]`, indicating the number of dimensions, can be distributed as a postfix to both the element type and the array name. Arrays of arrays are often called *multidimensional arrays*.

The following declarations are all equivalent:

Click here to view code image

```
int[][] mXnArray;      // two-dimensional array
int[]   mXnArray[];    // two-dimensional array
int     mXnArray[][];  // two-dimensional array
```

It is customary to combine the declaration with the construction of the multidimensional array.

Click here to view code image

```
int[][] mXnArray = new int[4][5];    // 4 x 5 matrix of ints
```

The previous declaration constructs an array `mXnArray` of four elements, where each element is an array (row) of five `int` values. The concept of rows and columns is often used to describe the dimensions of a two-dimensional array, which is often called a *matrix*. However, such an interpretation is not dictated by the Java language.

Each row in the previous matrix is denoted by `mXnArray[i]`, where `0 ≤ i < 4`. Each element in the `i`th row, `mXnArray[i]`, is accessed by `mXnArray[i][j]`, where `0 ≤ j < 5`. The number of rows is given by `mXnArray.length`, in this case `4`, and the number of values in the `i`th row is given by `mXnArray[i].length`, in this case `5` for all the rows, where `0 ≤ i < 4`.

Multidimensional arrays can also be constructed and explicitly initialized using the array initializers discussed for simple arrays. Note that each row is an array that uses an array initializer to specify its values:

Click here to view code image

```
double[][] identityMatrix = {
  {1.0, 0.0, 0.0, 0.0 }, // 1. row
  {0.0, 1.0, 0.0, 0.0 }, // 2. row
  {0.0, 0.0, 1.0, 0.0 }, // 3. row
  {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 floating-point matrix
```

Arrays in a multidimensional array need not have the same length; in which case, they are called *ragged arrays*. The array of arrays `pizzaGalore` in the following code has five rows; the first four rows have different lengths but the fifth row is left unconstructed:

Click here to view code image

```
Pizza[][] pizzaGalore = {
  { new Pizza(), null, new Pizza() },     // 1. row is an array of 3 elements.
  { null, new Pizza()},                   // 2. row is an array of 2 elements.
  new Pizza[1],                           // 3. row is an array of 1 element.
  {},                                     // 4. row is an array of 0 elements.
  null                                    // 5. row is not constructed.
};
```

When constructing multidimensional arrays with the `new` operator, the length of the deeply nested arrays may be omitted. In such a case, these arrays are left unconstructed. For example, an array of arrays to represent a room (defined by class `HotelRoom`) on a floor in a hotel on a street in a city can have the type `HotelRoom[][][][]`. From left to right, the square brackets represent indices for street, hotel, floor, and room, respectively. This four-dimensional array of arrays can be constructed piecemeal, starting with the leftmost dimension and proceeding to the rightmost successively.

Click here to view code image

```
HotelRoom[][][][] rooms = new HotelRoom[10][5][][];  // Just streets and hotel
```

The preceding declaration constructs the array of arrays `rooms` partially with 10 streets, where each street has five hotels. Floors and rooms can be added to a particular hotel on a particular street:

Click here to view code image

```
rooms[0][0]          = new HotelRoom[3][]; // 3 floors in 1st hotel on 1st street
rooms[0][0][0]       = new HotelRoom[8];   // 8 rooms on 1st floor in this hotel.
rooms[0][0][0][0]    = new HotelRoom();    // Initializes 1st room on this floor.
```

The next code snippet constructs an array of arrays `matrix`, where the first row has one element, the second row has two elements, and the third row has three elements. Note that the outer array is constructed first. The second dimension is constructed in a loop that constructs the array in each row. The elements in the multidimensional array will be implicitly initialized to the default `double` value (`0.0D`). In Figure 3.1, the array of arrays `matrix` is depicted after the elements have been explicitly initialized.

Click here to view code image

```
double[][] matrix = new double[3][];       // (1) Number of rows.

for (int i = 0; i < matrix.length; ++i)
  matrix[i] = new double[i + 1];           // Construct a row.
```

Figure 3.1 *Array of Arrays*

The type of the variable `matrix` is `double[][]` at (1), a two-dimensional array of `double` values. The type of the variable `matrix[i]` (where `0` ≤ `i`< `matrix.length`) is `double[]`, a one-dimensional array of `double` values. The type of the variable `matrix[i][j]` (where `0` ≤ `i`< `matrix.length` and `0` ≤ `j`< `matrix[i].length`) is `double`, a simple variable of type `double`.

Two other ways of initializing such an array of arrays are shown next. The first approach uses array initializers, and the second uses an anonymous array of arrays.

Click here to view code image

```
double[][] matrix2 = {     // (2) Using array initializers.
  {1.0},                   // 1. row
  {1.0, 2.0},              // 2. row
  {1.0, 2.0, 3.0}          // 3. row
};

double[][] matrix3 = new double[][] { // (3) Using an anonymous array of array
  {1.0},                   // 1. row
  {1.0, 2.0},              // 2. row
  {1.0, 2.0, 1.0}          // 3. row
};
```

Nested loops are a natural match for manipulating multidimensional arrays. In Example 3.9, a rectangular 4 × 3 `int` matrix is declared and constructed at (1). The program finds the minimum value in the matrix. The outer loop at (2) iterates over the rows (`mXnArray[i]`, where `0` ≤ `i`< `mXnArray.length`), and the inner loop at (3) iterates over the elements in each row in turn (`mXnArray[i][j]`, where `0` ≤ `j`< `mXnArray[i].length`). The outer loop is executed `mXnArray.length` times, or four times, and the inner loop is executed (`mXnArray.length`) × (`mXnArray[i].length`), or 12 times, since all rows have the same length 3.

The `for(:)` loop also provides a safe and convenient way of iterating over an array. Several examples of its use are provided in §4.8, p. 176.

Example 3.9 *Using Multidimensional Arrays*

Click here to view code image

```
public class MultiArrays {

  public static void main(String[] args) {
    // Declare and construct the M X N matrix.
    int[][] mXnArray = {                                              // (1)
        {16,  7, 12}, // 1. row
        { 9, 20, 18}, // 2. row
        {14, 11,  5}, // 3. row
        { 8,  5, 10}  // 4. row
    }; // 4 x 3 int matrix

    // Find the minimum value in an M X N matrix:
    int min = mXnArray[0][0];
    for (int i = 0; i < mXnArray.length; ++i)                      // (2)
      // Find min in mXnArray[i], in the row given by index i:
      for (int j = 0; j < mXnArray[i].length; ++j)                 // (3)
        min = Math.min(min, mXnArray[i][j]);

    System.out.println("Minimum value: " + min);
  }
}''
```

Output from the program:

```
Minimum value: 5
```

3.10 Parameter Passing

Objects communicate by calling methods on each other. A *method call* is used to invoke a method on an object. Parameters in the method call provide one way of exchanging information between the caller object and the callee object (which need not be different).

The syntax of a method call can be any one of the following:

Click here to view code image

object_reference`.`*method_name*`(`*actual_parameter_list*`)`

class_name`.`*static_method_name*`(`*actual_parameter_list*`)`

method_name`(`*actual_parameter_list*`)`

The *object_reference* must be an expression that evaluates to a reference value denoting the object on which the method is called. If the caller and the callee are the same, *object reference* can be omitted (see the discussion of the `this` reference on p. 106). The *class_name* can be the *fully qualified name* (§6.3, p. 326) of the class. The *actual_parameter_list* is *comma-separated* if there is more than one parameter. The parentheses are mandatory even if the actual parameter list is empty. This distinguishes the method call from field access. One can specify fully qualified names for classes and packages using the *dot operator* (`.`).

Click here to view code image

```
objRef.doIt(time, place);         // Explicit object reference
int i = java.lang.Math.abs(-1);   // Fully qualified class name
int j = Math.abs(-1);             // Simple class name
```

```
someMethod(ofValue);              // Object or class is implied
someObjRef.make().make().make();  // make() returns a reference value
```

The dot operator (`.`) has left associativity. In the last line of the preceding code, the first call of the `make()` method returns a reference value that denotes the object on which to execute the next call, and so on. This is an example of *call chaining*.

Each *actual parameter* (also called an *argument*) is an expression that is evaluated, and whose value is passed to the method when the method is invoked. Its value can vary from invocation to invocation. *Formal parameters* are parameters defined in the *method declaration* and are *local* to the method.

It should also be stressed that each invocation of a method has its own copies of the formal parameters, as is the case for any local variables in the method. The JVM uses a *stack* to keep track of method execution and a *heap* to manage the objects that are created by the program (§7.1, p. 365). Values of local variables and those passed to the method as parameters, together with any temporary values computed during the execution of the method, are always stored on the stack. Thus only primitive values and reference values are stored on the stack, and only these can be passed as parameters in a method call, but never any object from the heap.

In Java, all parameters are *passed by value*—that is, an actual parameter is evaluated and its value from the stack is assigned to the corresponding formal parameter. Table 3.2 summarizes the value that is passed depending on the type of the parameters. In the case of primitive data types, the data value of the actual parameter is passed. If the actual parameter is a reference to an object, the reference value of the denoted object is passed and not the object itself. Analogously, if the actual parameter is an array element of a primitive data type, its data value is passed, and if the array element is a reference to an object, then its reference value is passed.

Table 3.2 *Parameter Passing by Value*

Data type of the formal param	**Value passed**
Primitive data type	Primitive data value of the actual parameter
Reference type (i.e., class, interface, array, or enum type)	Reference value of the actual parameter

The order of evaluation in the actual parameter list is always *from left to right*. The evaluation of an actual parameter can be influenced by an earlier evaluation of an actual parameter. Given the following declaration:

```
int i = 4;
```

the method call

```
leftRight(i++, i);
```

is effectively the same as

```
leftRight(4, 5);
```

and not the same as

```
leftRight(4, 4);
```

An overview of the conversions that can take place in a method invocation context is provided in §2.4, p. 48. Method invocation conversions for primitive values are discussed in the next subsection (p. 129), and those for reference types are discussed in §5.10, p. 265. Calling variable arity methods is discussed in the next section (p. 136).

For the sake of simplicity, the examples in subsequent sections primarily show method invocation on the same object or the same class. The parameter passing mechanism is no different when different objects or classes are involved.

Passing Primitive Data Values

An actual parameter is an expression that is evaluated first, with the resulting value then being assigned to the corresponding formal parameter at method invocation. The use of this value in the method has no influence on the actual parameter. In particular, when the actual parameter is a variable of a primitive data type, the value of the variable from the stack is copied to the formal parameter at method invocation. Since formal parameters are local to the method, any changes made to the formal parameter will not be reflected in the actual parameter after the call completes.

Legal type conversions between actual parameters and formal parameters of *primitive data types* are summarized here from Table 2.17, p. 47:

- Primitive widening conversion
- Unboxing conversion, followed by an optional widening primitive conversion

These conversions are illustrated by invoking the following method

Click here to view code image

```
static void doIt(long i) { /* ... */ }
```

with the following code:

Click here to view code image

```
Integer intRef = 34;
Long longRef = 34L;
doIt(34);         // (1) Primitive widening conversion: long <-- int
doIt(longRef);    // (2) Unboxing: long <-- Long
doIt(intRef);     // (3) Unboxing, followed by primitive widening conversion:
                  //     long <-- int <-- Integer
```

However, for parameter passing, there are no implicit narrowing conversions for integer constant expressions (§2.4, p. 48).

Example 3.10 *Passing Primitive Values*

Click here to view code image

```
public class CustomerOne {
  public static void main (String[] args) {
    PizzaFactory pizzaHouse = new PizzaFactory();
    int pricePrPizza = 15;
    System.out.println("Value of pricePrPizza before call: " + pricePrPizza);
    double totPrice = pizzaHouse.calcPrice(4, pricePrPizza);             // (1
    System.out.println("Value of pricePrPizza after call: " + pricePrPizza);
  }
}

class PizzaFactory {
  public double calcPrice(int numberOfPizzas, double pizzaPrice) {       // (2
    pizzaPrice = pizzaPrice / 2.0;        // Changes price.
    System.out.println("Changed pizza price in the method: " + pizzaPrice);
    return numberOfPizzas * pizzaPrice;
  }
}
```

Output from the program:

Click here to view code image

```
Value of pricePrPizza before call: 15
Changed pizza price in the method: 7.5
Value of pricePrPizza after call: 15
```

In Example 3.10, the method `calcPrice()` is defined in the class `PizzaFactory` at (2). It is called from the `CustomerOne.main()` method at (1). The value of the first actual parameter, `4`, is copied to the `int` formal parameter `numberOfPizzas`. Note that the second actual parameter `pricePrPizza` is of the type `int`, while the corresponding formal parameter `pizzaPrice` is of the type `double`. Before the value of the actual parameter `pricePrPizza` is copied to the formal parameter `pizzaPrice`, it is implicitly widened to a `double`. The passing of primitive values is illustrated in Figure 3.2.

Figure 3.2 *Parameter Passing: Primitive Data Values*

The value of the formal parameter `pizzaPrice` is changed in the `calcPrice()` method, but this does not affect the value of the actual parameter `pricePrPizza` on return. It still has the value `15`. The bottom line is that the formal parameter is a local variable, and changing its value does not affect the value of the actual parameter.

Passing Reference Values

If the actual parameter expression evaluates to a reference value, the resulting reference value on the stack is assigned to the corresponding formal parameter reference at method invocation. In particular, if an actual parameter is a reference to an object, the reference value stored in the actual parameter is passed. Consequently, both the actual parameter and the formal parameter are aliases to the object denoted by this reference value during the invocation of the method. In particular, this implies that changes made to the object via the formal parameter *will* be apparent after the call returns.

Type conversions between actual and formal parameters of reference types are discussed in §5.10, p. 265.

In Example 3.11, a `Pizza` object is created at (1). Any object of the class `Pizza` created using the class declaration at (5) always results in a beef pizza. In the call to the `bake()` method at (2), the reference value of the object referenced by the actual parameter `favoritePizza` is assigned to the formal parameter `pizzaToBeBaked` in the declaration of the `bake()` method at (3).

Example 3.11 *Passing Reference Values*

Click here to view code image

```
public class CustomerTwo {
  public static void main (String[] args) {
    Pizza favoritePizza = new Pizza();              // (1)
    System.out.println("Meat on pizza before baking: " + favoritePizza.meat);
    bake(favoritePizza);                            // (2)
    System.out.println("Meat on pizza after baking: " + favoritePizza.meat);
  }
  public static void bake(Pizza pizzaToBeBaked) {   // (3)
    pizzaToBeBaked.meat = "chicken";  // Change the meat on the pizza.
    pizzaToBeBaked = null;                           // (4)
  }
}

class Pizza {                                        // (5)
  String meat = "beef";
}
```

Output from the program:

Click here to view code image

```
Meat on pizza before baking: beef
Meat on pizza after baking: chicken
```

One particular consequence of passing reference values to formal parameters is that any changes made to the object via formal parameters will be reflected back in the calling method when the call returns. In this case, the reference `favoritePizza` will show that chicken has been substituted for beef on the pizza. Setting the formal parameter `pizzaToBeBaked` to `null` at (4) does not change the reference value in the actual parameter `favoritePizza`. The situation at method invocation, and just before the return from method `bake()`, is illustrated in Figure 3.3.

Figure 3.3 *Parameter Passing: Reference Values*

In summary, the formal parameter can only change the *state* of the object whose reference value was passed to the method.

The parameter passing strategy in Java is *call by value* and not *call by reference*, regardless of the type of the parameter. Call by reference would have allowed values in the actual parameters to be changed via formal parameters; that is, the value in `pricePrPizza` would be halved in Example 3.10 and `favoritePizza` would be set to `null` in Example 3.11. However, this cannot be directly implemented in Java.

Passing Arrays

The discussion of passing reference values in the previous section is equally valid for arrays, as arrays are objects in Java. Method invocation conversions for array types are discussed along with those for other reference types in §5.10, p. 265.

In Example 3.12, the idea is to repeatedly swap neighboring elements in an integer array until the largest element in the array *percolates* to the last position in the array.

Example 3.12 *Passing Arrays*

Click here to view code image

```
public class Percolate {

  public static void main (String[] args) {
    int[] dataSeq = {8,4,6,2,1};    // Create and initialize an array.

    // Write array before percolation:
    printIntArray(dataSeq);

    // Percolate:
    for (int index = 1; index < dataSeq.length; ++index)

      if (dataSeq[index-1] > dataSeq[index])
        swap(dataSeq, index-1, index);                    // (1)

    // Write array after percolation:
    printIntArray(dataSeq);
  }

  public static void swap(int[] intArray, int i, int j) { // (2)
    int tmp = intArray[i]; intArray[i] = intArray[j]; intArray[j] = tmp;
  }

  public static void swap(int v1, int v2) {               // (3) Logical error
    int tmp = v1; v1 = v2; v2 = tmp;
  }

  public static void printIntArray(int[] array) {         // (4)
    for (int value : array)
      System.out.print(" " + value);
    System.out.println();
  }
}
```

Output from the program:

```
8 4 6 2 1
4 6 2 1 8
```

Note that in the declaration of the method `swap()` at (2), the formal parameter `intArray` is of the array type `int[]`. The other two parameters are of type `int`. They denote the values in the array that should be swapped. The signature of the method is

```
swap(int[], int, int)
```

This `swap()` method is called in the `main()` method at (1), where one of the actual parameters is the array variable `dataSeq`. The reference value of the array variable `dataSeq` is assigned to the array variable `intArray` at method invocation. After return from the call to the `swap()` method, the array variable `dataSeq` will reflect the changes made to the array via the corresponding formal parameter. This situation is depicted in Figure 3.4 at the first call and return from the `swap()` method, indicating how the values of the elements at indices `0` and `1` in the array have been swapped.

Figure 3.4 *Parameter Passing: Arrays*

However, the declaration of the `swap()` method at (3) will *not* swap two values. The method call

Click here to view code image

```
swap(dataSeq[index-1], dataSeq[index]); // Call signature: swap(int, int)
```

will result in the `swap()` method at (3) to be invoked. Its execution will have no effect on the array elements, as the swapping is done on the values of the formal parameters.

The method `printIntArray()` at (4) also has a formal parameter of array type `int[]`. Note that the formal parameter is specified as an array reference using the `[]` notation, but this notation is not used when an array is passed as an actual parameter.

Array Elements as Actual Parameters

Array elements, like other variables, can store values of primitive data types or reference values of objects. In the latter case, they can also be arrays—that is, arrays of arrays (p. 124). If an array element is of a primitive data type, its data value is passed; if it is a reference to an object, the reference value is passed. The method invocation conversions apply to the values of array elements as well.

Example 3.13 *Array Elements as Primitive Data Values*

Click here to view code image

```
public class FindMinimum {

  public static void main(String[] args) {
    int[] dataSeq = {6,4,8,2,1};

    int minValue = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      minValue = minimum(minValue, dataSeq[index]);            // (1)

    System.out.println("Minimum value: " + minValue);
  }

  public static int minimum(int i, int j) {                     // (2)
    return (i <= j) ? i : j;
  }
}
```

Output from the program:

```
Minimum value: 1
```

In Example 3.13, the value of all but one element of the array `dataSeq` is retrieved and passed consecutively at (1) to the formal parameter `j` of the `minimum()` method defined at (2). The discussion on passing primitive values (p. 129) also applies to array elements that have primitive values.

In Example 3.14, the formal parameter `seq` of the `findMinimum()` method defined at (4) is an array variable. The variable `matrix` denotes an array of arrays declared at (1) simulating a multidimensional array that has three rows, where each row is a simple array. The first row, denoted by `matrix[0]`, is passed to the `findMinimum()` method in the call at (2). Each remaining row is passed by its reference value in the call to the `findMinimum()` method at (3).

Example 3.14 *Array Elements as Reference Values*

Click here to view code image

```
public class FindMinimumMxN {

  public static void main(String[] args) {
    int[][] matrix = { {8,4},{6,3,2},{7} };                    // (1)

    int min = findMinimum(matrix[0]);                          // (2)
    for (int i = 1; i < matrix.length; ++i) {
      int minInRow = findMinimum(matrix[i]);                   // (3)
      min = Math.min(min, minInRow);
    }
    System.out.println("Minimum value in matrix: " + min);
  }

  public static int findMinimum(int[] seq) {                   // (4)
    int min = seq[0];
    for (int i = 1; i < seq.length; ++i)
      min = Math.min(min, seq[i]);
    return min;
  }
}
```

Output from the program:

```
Minimum value in matrix: 2
```

`final` Parameters

A formal parameter can be declared with the keyword `final` preceding the parameter declaration in the method declaration. A `final` parameter is also known as a *blank final variable*; that is, it is blank (uninitialized) until a value is assigned to it, (e.g., at method invocation) and then the value in the variable cannot be changed during the lifetime of the variable (see also the discussion in §6.6, p. 352). The compiler can treat `final` variables as constants for code optimization purposes. Declaring parameters as `final` prevents their values from being changed inadvertently. A formal parameter's declaration as `final` does not affect the caller's code.

The declaration of the method `calcPrice()` from Example 3.10 is shown next, with the formal parameter `pizzaPrice` declared as `final`:

Click here to view code image

```
public double calcPrice(int numberOfPizzas, final double pizzaPrice) {  // (2'
  pizzaPrice = pizzaPrice/2.0;         // (3) Not allowed. Compile-time error!
  return numberOfPizzas * pizzaPrice;
}
```

If this declaration of the `calcPrice()` method is compiled, the compiler will not allow the value of the `final` parameter `pizzaPrice` to be changed at (3) in the body of the method.

As another example, the declaration of the method `bake()` from Example 3.11 is shown here, with the formal parameter `pizzaToBeBaked` declared as `final`:

Click here to view code image

```
public static void bake(final Pizza pizzaToBeBaked) { // (3)
  pizzaToBeBaked.meat = "chicken";    // (3a) Allowed
  pizzaToBeBaked = null;              // (4) Not allowed. Compile-time error!
}
```

If this declaration of the `bake()` method is compiled, the compiler will not allow the reference value of the `final` parameter `pizzaToBeBaked` to be changed at (4) in the body of the method. Note that this applies to the reference value in the `final` parameter, but not to the object denoted by this parameter. The state of the object can be changed as before, as shown at (3a).

For use of the `final` keyword in other contexts, see §5.5, p. 225.

3.11 Variable Arity Methods

A *fixed arity* method must be called with the same number of actual parameters (also called *arguments*) as the number of formal parameters specified in its declaration. If the method declaration specifies two formal parameters, every call of this method must specify exactly two arguments. We say that the arity of this method is 2. In other words, the arity of such a method is fixed, and it is equal to the number of formal parameters specified in the method declaration.

Java also allows declaration of *variable arity* methods (also called *varargs* methods), meaning that the number of arguments in its call can be *varied*. As we shall see, invocations of such a method may contain more actual parameters than formal parameters. Variable arity methods are heavily employed in formatting text representation of values, as demonstrated by the variable arity method `System.out.printf()` that is used in many examples for this purpose.

The *last* formal parameter in a variable arity method declaration is declared as follows:

Click here to view code image

```
type... formal_parameter_name
```

The ellipsis (`...`) is specified between the *type* and the *formal_parameter_name*. The *type* can be a primitive type, a reference type, or a type parameter. Whitespace can be specified on both sides of the ellipsis. Such a parameter is usually called a *variable arity parameter* (also known as a *varargs* parameter).

Apart from the variable arity parameter, a variable arity method is identical to a fixed arity method. The method `publish()` below is a variable arity method:

Click here to view code image

```
public static void publish(int n, String... data) {       // (int, String[])
  System.out.println("n: " + n + ", data size: " + data.length);
}
```

The variable arity parameter in a variable arity method is always interpreted as having an array type:

```
type[]
```

In the body of the `publish()` method, the variable arity parameter `data` has the type `String[]`, so it is a simple array of `String`s.

Only *one* variable arity parameter is permitted in the formal parameter list, and it is always the *last* parameter in the list. Given that the method declaration has *n* formal parameters and the method call has *k* actual parameters, *k* must be equal to or greater than *n* – 1. The last *k* – *n* + 1 actual parameters are evaluated and stored in an array whose reference value is passed as the value of the actual parameter. In the case of the `publish()` method, *n* is equal to 2, so *k* can be 1, 2, 3, and so on. The following invocations of the `publish()` method show which arguments are passed in each method call:

Click here to view code image

```
publish(1);                  // (1, new String[] {})
publish(2, "two");           // (2, new String[] {"two"})
publish(3, "two", "three");  // (3, new String[] {"two", "three"})
```

Each method call results in an *implicit* array being created and passed as an argument. This array can contain zero or more argument values that do *not* correspond to the formal parameters preceding the variable arity parameter. This array is referenced by the variable arity parameter `data` in the method declaration. The preceding calls would result in the `publish()` method printing the following output:

```
n: 1, data size: 0
n: 2, data size: 1
n: 3, data size: 2
```

To overload a variable arity method, it is not enough to change the type of the variable arity parameter to an explicit array type. The compiler will complain if an attempt is made to overload the method `transmit()`, as shown in the following code:

Click here to view code image

```
public static void transmit(String... data) {  }  // Compile-time error!
public static void transmit(String[] data)  {  }  // Compile-time error!
```

Both methods above have the signature `transmit(String[])`. These declarations would result in two methods with equivalent signatures in the same class, which is not permitted.

Overloading and overriding of methods with variable arity are discussed in §5.10, p. 265.

Calling a Variable Arity Method

Example 3.15 illustrates various aspects of calling a variable arity method. The method `flexiPrint()` in the `VarargsDemo` class has a variable arity parameter:

Click here to view code image

```
public static void flexiPrint(Object... data) { // Object[]
  //...
}
```

The variable arity method prints the name of the `Class` object representing the *actual array* that is passed at runtime. It prints the number of elements in this array as well as the text representation of each element in the array.

The method `flexiPrint()` is called in the `main()` method. First it is called with the values of primitive types and `String`s ((1) to (8)), and then it is called with the program arguments (p. 141) supplied on the command line ((9) to (11)).

Compiling the program results in a *warning* at (9), which we ignore for the time being. The program can still be run, as shown in Example 3.15. The numbers at the end of the lines in the output relate to numbers in the code, and are not printed by the program.

Example 3.15 *Calling a Variable Arity Method*

Click here to view code image

```
public class VarargsDemo {
  public static void flexiPrint(Object... data) { // Object[]
    // Print the name of the Class object for the varargs parameter.
    System.out.print("Type: " + data.getClass().getName());

    System.out.println("  No. of elements: " + data.length);

    System.out.print("Element values: ");
    for(Object element : data)
      System.out.print(element + " ");
    System.out.println();
  }

  public static void main(String... args) {
    int    day       = 13;
    String monthName = "August";
    int    year      = 2009;

    // Passing primitives and non-array types:
    flexiPrint();                      // (1) new Object[] {}
    flexiPrint(day);                   // (2) new Object[] {Integer.valueOf(da
    flexiPrint(day, monthName);        // (3) new Object[] {Integer.valueOf(da
                                       //                   monthName}
    flexiPrint(day, monthName, year);  // (4) new Object[] {Integer.valueOf(da
                                       //                   monthName,
                                       //                   Integer.valueOf(ye
    System.out.println();

    // Passing an array type:
    Object[] dateInfo = {day,          // (5) new Object[] {Integer.valueOf(da
                         monthName,    //                   monthName,
                         year};        //                   Integer.valueOf(ye
    flexiPrint(dateInfo);              // (6) Non-varargs call
    flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
    flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
    System.out.println();
    // Explicit varargs or non-varargs call:
    flexiPrint(args);                  // (9) Warning!
    flexiPrint((Object) args);         // (10) Explicit varargs call
    flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
  }
}
```

Compiling the program:

Click here to view code image

```
>javac VarargsDemo.java
VarargsDemo.java:41: warning: non-varargs call of varargs method with inexact
argument type for last parameter;
    flexiPrint(args);                  // (9) Warning!
               ^
  cast to Object for a varargs call
  cast to Object[] for a non-varargs call and to suppress this warning
1 warning
```

Running the program:

Click here to view code image

```
>java VarargsDemo To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 0                 (1)
Element values:
Type: [Ljava.lang.Object;  No. of elements: 1                 (2)
Element values: 13
Type: [Ljava.lang.Object;  No. of elements: 2                 (3)
Element values: 13 August
Type: [Ljava.lang.Object;  No. of elements: 3                 (4)
Element values: 13 August 2009

Type: [Ljava.lang.Object;  No. of elements: 3                 (6)
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 1                 (7)
Element values: [Ljava.lang.Object;@1eed786
Type: [Ljava.lang.Object;  No. of elements: 1                 (8)
Element values: [Ljava.lang.Object;@1eed786

Type: [Ljava.lang.String;  No. of elements: 6                 (9)
Element values: To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 1                 (10)
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6                 (11)
Element values: To arg or not to arg
```

Variable Arity and Fixed Arity Method Calls

The calls at (1) to (4) in Example 3.15 are all *variable arity calls*, as an implicit `Object` array is created in which the values of the actual parameters are stored. The reference value of this array is passed to the method. The printout shows that the type of the parameter is actually an array of `Object`s (`[Ljava.lang.Object;`).

The call at (6) differs from the previous calls in that the actual parameter is an array that has the *same* type (`Object[]`) as the variable arity parameter, without having to create an implicit array. In such a case, *no* implicit array is created, and the reference value of the array `dateInfo` is passed to the method. See also the result from this call at (6) in the output. The call at (6) is a *fixed arity call* (also called a *non-varargs call*), where no implicit array is created:

Click here to view code image

```
flexiPrint(dateInfo);              // (6) Non-varargs call
```

However, if the actual parameter is cast to the type `Object` as at (7), a *variable arity* call is executed:

Click here to view code image

```
flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
```

The type of the actual argument (`Object`) is now *not* the same as that of the variable arity parameter (`Object[]`), resulting in an array of the type `Object[]` being created in which the array `dateInfo` is stored as an element. The printout at (7) shows that only the text representation of the `dateInfo` array is printed, and not its elements, as it is the sole element of the implicit array.

The call at (8) is a *fixed arity* call, for the same reason as the call at (6). Now, however, the array `dateInfo` is explicitly stored as an element in an array of the type `Object[]` that matches the type of the variable arity parameter:

Click here to view code image

```
flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
```

The output from (8) is the same as the output from (7), where the array `dateInfo` was passed as an element in an implicitly created array of type `Object[]`.

The compiler issues a *warning* for the call at (9):

Click here to view code image

```
flexiPrint(args);                  // (9) Warning!
```

The actual parameter `args` is an array of the type `String[]`, which is a *subtype* of `Object[]`—the type of the variable arity parameter. The array `args` can be passed in a fixed arity call as an array of the type `String[]`, or in a variable arity call as *an element* in an implicitly created array of the type `Object[]`. *Both* calls are feasible and valid in this case. Note that the compiler chooses a fixed arity call rather than a variable arity call, but also issues a warning. The result at (9) confirms this course of action. A warning at compile time is not the same as a compile-time error. The former does not prevent the program from being run, whereas the latter does.

At (10), the array `args` of the type `String[]` is explicitly passed as an `Object` in a variable arity call, similar to the call at (7):

Click here to view code image

```
flexiPrint((Object) args);          // (10) Explicit varargs call
```

At (11), the array `args` of type `String[]` is explicitly passed as an array of the type `Object[]` in a fixed arity call. This call is equivalent to the call at (9), where the widening reference conversion is implicit, but now without a warning at compile time. The two calls print the same information, as is evident from the output at (9) and (11):

Click here to view code image

```
flexiPrint((Object[]) args);        // (11) Explicit non-varargs call
```

3.12 The `main()` Method

The mechanics of compiling and running Java applications using the JDK are outlined in §1.8, p. 19. The `java` command executes a method called `main` in the class specified on the command line. This class designates the *entry point of the application*. Any class can have a `main()` method, but only the `main()` method of the class specified in the `java` command starts the execution of a Java application.

The `main()` method must have `public` access so that the JVM can call this method (§6.5, p. 345). It is a `static` method belonging to the class, so no object of the class is required to start its execution. It does not return a value; that is, it is declared as `void`. It has an array of `String` objects as its only formal parameter. This array contains any arguments passed to the program on the command line (see the next subsection). The following method header declarations fit the bill, and any one of them can be used for the `main()` method:

Click here to view code image

```
public static void main(String[] args)     // Method header
public static void main(String args[])     // Method header
public static void main(String... args)    // Method header
```

The three modifiers can occur in any order in the method header. The requirements given in these examples do not exclude specification of other non-access modifiers like `final` (§5.5, p. 226) or a `throws` clause (§7.5, p. 388). The `main()` method can also be overloaded like any other method. The JVM ensures that the `main()` method having the correct method header is the starting point of program execution.

Program Arguments

Any arguments passed to the program on the command line can be accessed in the `main()` method of the class specified on the command line. These arguments are passed to the `main()` method via its formal parameter `args` of type `String[]`. These arguments are called *program arguments*.

In Example 3.16, the program prints the arguments passed to the `main()` method from the following command line:

Click here to view code image

```
>java Colors red yellow green "blue velvet"
```

The program prints the total number of arguments given by the field `length` of the `String` array `args`. Each string in `args`, which corresponds to a program argument, is printed together with its length inside a `for` loop. From the output, we see that there are four program arguments. On the command line, the arguments can be separated by one or more spaces between them, but these are not part of any argument. The last argument shows that we can quote the argument if spaces are to be included as part of the argument.

When no arguments are specified on the command line, a `String` array of zero length is created and passed to the `main()` method. Thus the reference value of the formal parameter in the `main()` method is never `null`.

Note that the command name `java` and the class name `Colors` are not passed to the `main()` method of the class `Colors`, nor are any other options that are specified on the command line.

As program arguments can only be passed as strings, they must be explicitly converted to other values by the program, if necessary.

Program arguments supply information to the application, which can be used to tailor the runtime behavior of the application according to user requirements.

Example 3.16 *Passing Program Arguments*

Click here to view code image

```
public class Colors {
   synchronized public static void main(String[] args) {
    System.out.println("No. of program arguments: " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println("Argument no. " + i + " (" + args[i] + ") has " +
                          args[i].length() + " characters.");
  }
}
```

Running the program:

Click here to view code image

```
>java Colors red yellow green "blue velvet"
No. of program arguments: 4
Argument no. 0 (red) has 3 characters.
Argument no. 1 (yellow) has 6 characters.
Argument no. 2 (green) has 5 characters.
Argument no. 3 (blue velvet) has 11 characters.
```

3.13 Local Variable Type Inference

A variable declaration requires the *type* of the variable to be specified in the declaration. However, in the case of *local variables*, the type can be specified by the reserved type name `var`, if the local declaration also specifies an *initialization expression* in the declaration. The compiler uses the type of the initialization expression to *infer* the type of the local variable. The restricted type name `var` denotes this inferred type in the local declaration. This is an example of *type inference*, where the type of a variable or an expression is derived from the context in which it is used. If the compiler cannot infer the type, it reports a compile-time error. A local variable declaration that uses `var` is also called a `var` *declaration*. A local variable declared this way is no different from any other local variable.

It is important to note that the type of the local variable is *solely* inferred from the initialization expression specified in the declaration. The following variable declaration in a local context (e.g., body of a method) is declared using the reserved type name `var`:

```
var year = 2022;
```

The compiler is able to infer that the type of the initialization expression `2022` is `int` in the above declaration, and therefore the variable `year` has the type `int`. The declaration above is equivalent to the declaration below, where the type is explicitly specified:

```
int year = 2022;
```

A cautionary note going forward: This subsection refers to many concepts and constructs that might not be familiar at this stage. It might be a good idea to get an overview now and to come back later for a more thorough review of this topic. The exhaustive index at the end of the book can of course be used at any time to look up a topic.

The class `ValidLVTI` in Example 3.17 illustrates valid uses of the restricted type name `var`. The comments in the code should be self-explanatory.

The `var` restricted type name is allowed in local variable declarations in *blocks* (including *initializer blocks*), *constructors*, and *methods*, as can be seen in the class `ValidLVTI` at (1a), (1b), and (2) and the method `main()`, respectively.

Note that at (3b) and (3c), the compiler is able to infer the type of the local variable from the return type of the method on the right-hand side.

It is worth noting that the *cast operator*, `()`, can be necessary to indicate the desired type, as shown at (5) and (7).

For array variables, the initialization expression must be an *array creation expression* that allows the array size and the array element type to be inferred, as shown at (11a), (11b), (11c), and (11d). A local declaration with `var` requires an initialization expression, which in the case of local arrays must be either an array creation expression or an anonymous array expression. In other words, it should be possible to infer both the array element type and the size of the array. It cannot be an array initializer.

The bodies (and the headers) of the `for(;;)` and `for(:)` *loops* can define their own local variables in their *block scope*. The type of the local variable `vowel` at (13) is inferred to be `char` from the array `vowels` (of type `char[]`) in the header of the `for(:)` loop. The type of the local variable `i` in the header of the `for(;;)` loop at (16) is determined to be `int` from the initial value. The `switch` statement also defines its own block scope in which local variables can be declared, as shown at (18).

Example 3.17 *Illustrating Local Variable Type Reference*

Click here to view code image

```
// Class ValidLVTI illustrates valid use of the restricted type name var.
public class ValidLVTI {

  // Static initializer block:
  static {
    var slogan = "Keep calm and code Java.";        // (1a) Allowed in static
  }                                                 //      initializer block

  // Instance initializer block:
  {
    var banner = "Keep calm and catch exceptions."; // (1b) Allowed in instanc
  }                                                 //      initializer block

  // Constructor:
  public ValidLVTI() {
    var luckyNumber = 13;                       // (2) Allowed in a constructo
  }

  // Method:
  public static void main(String[] args) {

    var virus = "COVID-19";                     // (3a) Type of virus is Strin
    var acronym = virus.substring(0, 5);        // (3b) Type of acronym is Str
    var num = Integer.parseInt(virus.substring(6)); // (3c) Type of num is int

    var obj = new Object();                     // (4) Type of obj is Object.
    var title = (String) null; // (5) Initialization expression type is String
                               //     Type of title is String.
    var sqrtOfNumber = Math.sqrt(100); // (6) Type of sqrtOfNumber is double,
                                       //     since the method returns
                                       //     a double value.

    var tvSize  = (short) 55;  // (7) Type of tvSize is short.
    var tvSize2 = 65;          // (8) Type of tvSize2 is int.

    var diameter = 10.0;       // (9) Type of diameter is double.
    var radius = 2.5F;         // (10) Type of radius is float.

    // Arrays:
    var vowels = new char[] {'a', 'e', 'i', 'o', 'u' }; // (11a) Type of vowel
                                                        // is char[]. Size is
    var zodiacSigns = new String[12]; // (11b) Type of zodiacSigns is String[]
                                      //       Size is 12.
    var a_2x3 = new int[2][3]; // (11c) Type of a_2x3 is int[][]. Size is 2x3.
    var a_2xn = new int[2][];  // (11d) Type of a_2xn is int[][]. Size is 2x?,
                               //       where second dimension can be undefine

    // The for(:) loop:
    var word1 = "";            // (12) Type of word2 is String.
    for (var vowel : vowels) { // (13) Type of vowel is char in the for(:)loop
      var letter = vowel;      // (14) Type of letter is char.
      word1 += letter;
    }

    // The for(;;) loop:
    var word2 = "";                          // (15) Type of word2 is String.
    for (var i = 0; i < vowels.length; i++) { // (16) Type of i is int in
```

```
                                              //      the for loop.
      var letter = vowels[i];                 // (17) Type of letter is char.
      word2 += letter;
    }

    // switch-statement:
    switch(virus) {
      case "Covid-19":
        var flag = "Needs to be tested.";     // (18) Type is String.
        // Do testing.
        break;
      default: // Do nothing.
    }
  }
}
```

Click here to view code image

```
// Class InvalidLVTI illustrates invalid use of the restricted type name var.
public class InvalidLVTI {

  var javaVendor = "Oracle"; // (19) Not allowed in instance variable declarat

  static var javaVersion = 11; // (20) Not allowed in static variable declarat

  public static void main(var args) { // (21) Not allowed for method parameter

    var name;              // (22) Not allowed without initialization expressi

    var objRef = null;     // (23) Literal null not allowed.

    var x = 10.0, y = 20.0, z = 40;   // (24) Not allowed in compound declarat

    var vowelsOnly = {'a', 'e', 'i', 'o', 'u' }; // (25) Array initializer not
                                                 //      allowed.
    var attendance = new int[];         // (26) Non-empty dimension required.
    var array3Dim = new String[][2][];  // (27) Cannot specify an empty dimens
                                        //      before a non-empty dimension.
    var letters[] = new char[]{'a', 'e', 'i', 'o', 'u' }; // (28) var not allo
                                                          //      as element t

    var prompt = prompt + 1;            // (29) Self-reference not allowed in
                                        //      initialization expression.
  }

  public static var getPlatformName() { // (30) Not allowed as return type.
    return "JDK";
  }
}
```

The following examples of invalid uses of the restricted type name `var` are shown in the class `InvalidLVTI` in Example 3.17:

- *Not allowed in field variable declarations*

  The `var` restricted type name is not allowed in field variable declarations, as shown at (19) and (20).

- *Not allowed in declaring formal parameters*

  Formal parameters in methods and constructors cannot be declared with `var`, as shown at (21) for the parameter `args` in the `main()` method.

- *Initialization expression is mandatory*

The `var` restricted type name is not allowed in a local variable declaration if an initialization expression is not specified, as shown at (22).

- *Initialization expression cannot be the* `null` *literal value*

  Since the literal `null` can be assigned to any reference type, a specific type for `objRef` at (23) cannot be determined. At (5), the cast `(String)` specifies the type of the initialization expression.

- *Cannot use* `var` *in compound declarations*

  The reserved type name `var` cannot be used in a *compound declaration*—that is, a declaration that declares several variables, as shown at (24).

- *Cannot use* `var` *when an array initializer is specified*

  As shown at (25), an *array initializer* cannot be used in a `var` declaration. However, an *array initialization expression* is allowed, as at (11a).

- *Array creation expression must specify the size*

  As in the case when an explicit type is specified for an array variable, the array creation expressions in the declaration must also specify the array size when using `var`; otherwise, the compiler will issue an error, as at (26) and (27). Valid array creation expressions specifying correct size are shown at (11b), (11c), and (11d).

- *Cannot use* `var` *as an array element type*

  The square brackets (`[]`) on the left-hand side at (28) are not allowed, as they indicate that the local variable is an array. Array type and size are solely determined from the initialization expression, as at (11a), (11b), (11c), and (11d).

- *Cannot have a self-reference in an initialization expression*

  As in the case when an explicit type is specified for the local variable, the initialization expression cannot refer to the local variable being declared, as at (29), where the variable is not initialized before use.

- *Cannot use* `var` *as the return type of a method*

  The method declaration at (30) cannot specify the return type using `var`.

- *A type cannot be a named* `var`

  As `var` is a reserved type name, it is *not* a valid name for a reference type; that is, a class, an interface, or an enum cannot be named `var`. In other contexts, it can be used as an identifier, but this is not recommended.

  Click here to view code image

  ```
  public class var {}   // var is not permitted as a class name. Compile-t
  ```

The reserved type name `var` should be used judiciously as the code can become difficult to understand. When reading the local declaration below, the initialization expression does not divulge any information about the type, and the names are not too helpful:

```
var x = gizmo.get();
```

Unless it is intuitively obvious, a human reader will have to resort to the API documentation in order to infer the type. Using intuitive names becomes even more important when using the reserved type name `var`.

We will revisit the restricted type name `var` when discussing *exception handling with try-with-resources* (§7.7, p. 407), using *generics* in local variable declarations (§11.2, p. 571), and specifying *inferred-type lambda parameters* (§13.2, p. 680).

Review Questions

3.7 How many of the following array declaration statements are legal?

Click here to view code image

```
int []aa[] = new int [4][4];
int bb[][] = new int [4][4];
int cc[][] = new int [][4];
int []dd[] = new int [4][];
int [][]ee = new int [4][4];
```

Select the one correct answer.

a. 0

b. 1

c. 2

d. 3

e. 4

f. 5

3.8 Which of these array declaration statements are legal? Select the three correct answers.

a. `int[] i[] = { { 1, 2 }, { 1 }, {}, { 1, 2, 3 } };`

b. `int i[] = new int[2] {1, 2};`

c. `int i[][] = new int[][] { {1, 2, 3}, {4, 5, 6} };`

d. `int[][] i = { { 1, 2 }, new int[2] };`

e. `int i[4] = { 1, 2, 3, 4 };`

3.9 What would be the result of compiling and running the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    int size = 20;
    int[] arr = new int[ size ];
    for (int i = 0; i < size; ++i) {
      System.out.println(arr[i]);
    }
  }
}
```

Select the one correct answer.

a. The code will fail to compile because the array type `int[]` is incorrect.

b. The program will compile, but it will throw an `ArrayIndexOutOfBoundsException` when run.

c. The program will compile and run without error, but it will produce no output.

d. The program will compile and run without error and will print the numbers 0 through 19.

e. The program will compile and run without error and will print `0` twenty times.

f. The program will compile and run without error and will print `null` twenty times.

3.10 What would be the result of compiling and running the following program?

Click here to view code image

```
public class DefaultValuesTest {
  int[] ia = new int[1];
  boolean b;
  int i;
  Object o;
  public static void main(String[] args) {
    DefaultValuesTest instance = new DefaultValuesTest();
    instance.print();
  }

  public void print() {
    System.out.println(ia[0] + " " + b + " " + i + " " + o);
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of uninitialized variables.

b. The program will throw a `java.lang.NullPointerException` when run.

c. The program will print `0 false NaN null`.

d. The program will print `0 false 0 null`.

e. The program will print `null 0 0 null`.

f. The program will print `null false 0 null`.

3.11 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class ParameterPass {
  public static void main(String[] args) {
    int i = 0;
    addTwo(i++);
    System.out.println(i);
  }
  static void addTwo(int i) {
    i += 2;
  }
}
```

Select the one correct answer.

a. 0

b. 1

c. 2

d. 3

3.12 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Passing {
  public static void main(String[] args) {
    int a = 0; int b = 0;
    int[] bArr = new int[1]; bArr[0] = b;

    inc1(a); inc2(bArr);

    System.out.println("a=" + a + " b=" + b + " bArr[0]=" + bArr[0]);
    }

    public static void inc1(int x) { x++; }

    public static void inc2(int[] x) { x[0]++; }
  }
```

Select the one correct answer.

a. The code will fail to compile, since `x[0]++;` is not a legal statement.

b. The code will compile and will print `a=1 b=1 bArr[0]=1` at runtime.

c. The code will compile and will print `a=0 b=1 bArr[0]=1` at runtime.

d. The code will compile and will print `a=0 b=0 bArr[0]=1` at runtime.

e. The code will compile and will print `a=0 b=0 bArr[0]=0` at runtime.

3.13 Given the following code:

Click here to view code image

```
public class RQ810A40 {
  static void print(Object... obj) {
    System.out.println("Object...: " + obj[0]);
  }
  public static void main(String[] args) {
    // (1) INSERT METHOD CALL HERE
  }
}
```

Which method call, when inserted at (1), will not result in the following output from the program?

```
Object...: 9
```

Select the one correct answer.

a. `print("9", "1", "1");`

b. `print(9, 1, 1);`

c. `print(new int[] {9, 1, 1});`

d. `print(new Integer[] {9, 1, 1});`

e. `print(new String[] {"9", "1", "1"});`

f. `print(new Object[] {"9", "1", "1"});`

g. None of the above

3.14 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  public static int add(int x, int y)  {
    var z = x + y;      // (1)
```

```
      return z;
    }
    public static void main(String[] args) {
      var a = 2, b = 3;   // (2)
      var z = add(a,b);   // (3)
    }
  }
```

Select the one correct answer.

a. Line (1) will fail to compile.
b. Line (2) will fail to compile.
c. Line (3) will fail to compile.
d. The code will compile successfully.

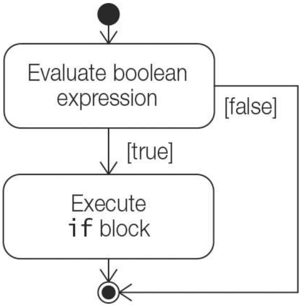
3.15 Given the following code:

Click here to view code image

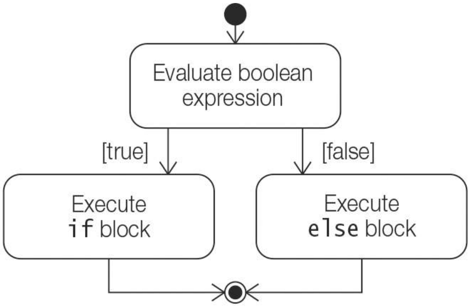
```
  public class TestMe {
    /* (1) INSERT METHOD HEADER HERE */ {
      return (double)x / y;
    }
    public static void main(String[] args) {
      double x = divide(2, 3);
    }
  }
```

Which method headers, when inserted individually at (1), will allow the code to compile?
Select the three correct answers.

a. `public static var divide(double x, double y)`
b. `public static double divide(double x, var y)`
c. `public static double divide(var x, double y)`
d. `public static double divide(int x, int y)`
e. `public static double divide(int x, double y)`
f. `public static double divide(double x, int y)`

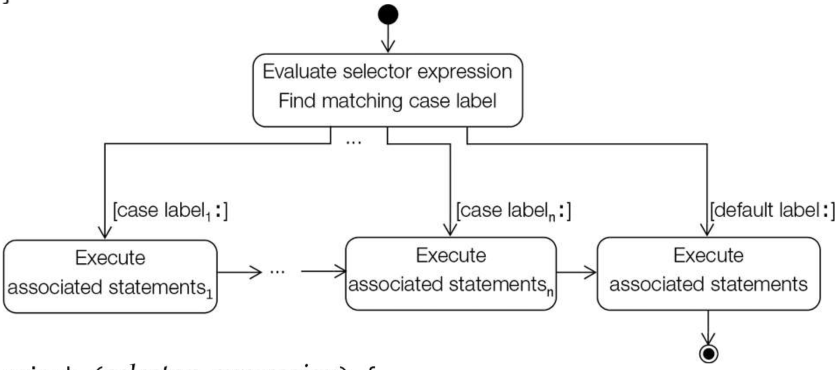

Evaluate boolean expression
[false]
[true]
Execute if block

(a) The simple `if` statement

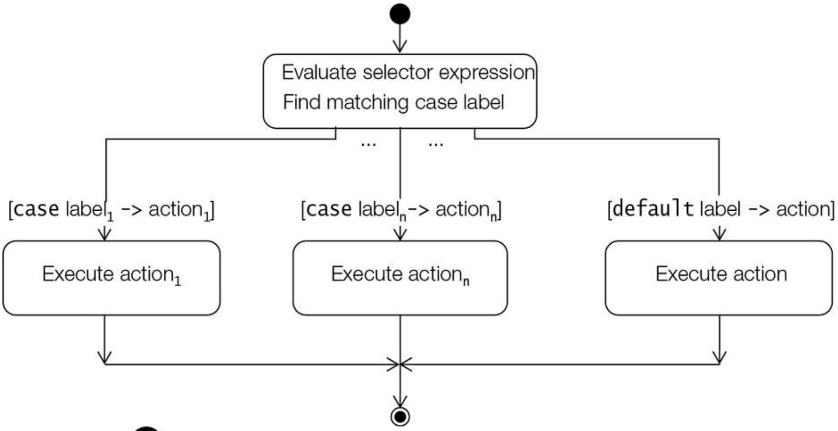

Evaluate boolean expression
[true]
[false]
Execute if block
Execute else block

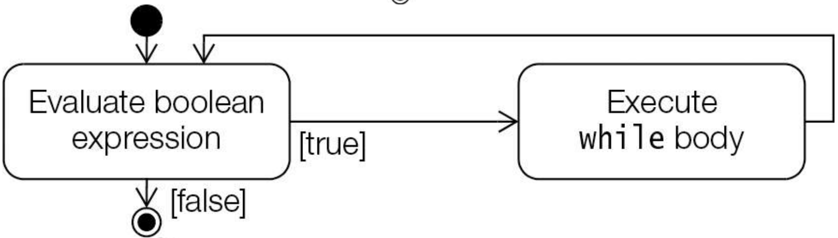
(b) The `if-else` statement

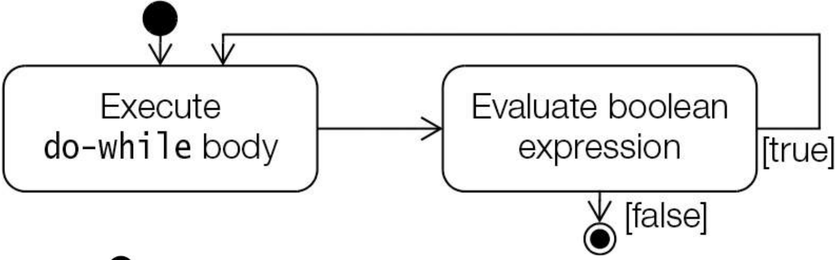
```
switch (selector_expression) {
  // Switch block with statement groups defined using colon notation:
  case CC:                                  statements
  case CC1: case CC2: ... case CCn:  statements
  case CC3, CC4, ..., CCm:                 statements
  ...
  default: ...
}
```

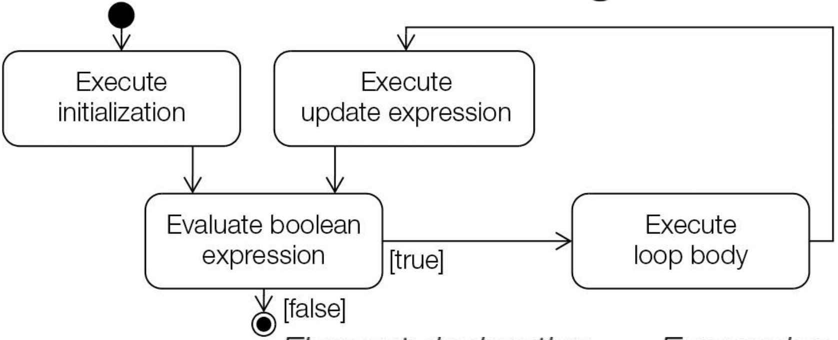

Evaluate selector expression
Find matching case label
...
[case label1:]
[case labeln:]
[default label:]
Execute associated statements1
...
Execute associated statementsn
Execute associated statements

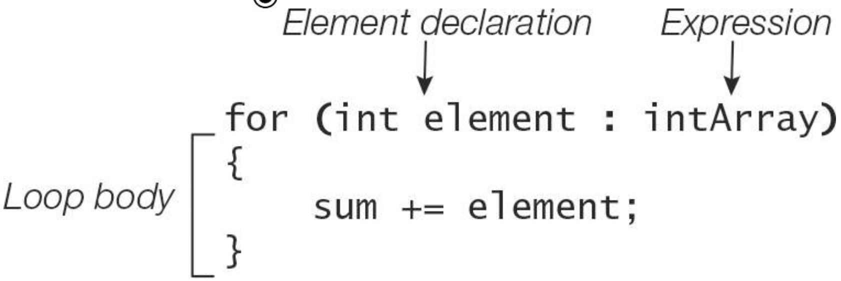
```
switch (selector_expression) {
  // Switch block with switch rules defined using arrow notation:
  case CC                   -> expression_statement;
  case CC1, CC2, ..., CCm  -> block
  case CC4                  -> throw_statement
  ...
  default                   -> ...
}
```


Evaluate selector expression
Find matching case label
...
...
[case label$_1$ -> action$_1$]
[case label$_n$-> action$_n$]
[default label -> action]
Execute action$_1$
Execute action$_n$
Execute action


Evaluate boolean expression
[true]
Execute while body
[false]


Execute do-while body
Evaluate boolean expression
[true]
[false]


Execute initialization
Execute update expression
Evaluate boolean expression
[true]
Execute loop body
[false]


Element declaration
Expression
Loop body

```
for (int element : intArray)
{
    sum += element;
}
```

Control Flow 4

Chapter Topics

- Choosing between alternative actions with the selection statements: the `if` statement, the `if-else` statement, the `switch` statement, and the `switch` expression
- Repeatedly executing code with the iteration statements: the `for` loop (`for(;;)`), the enhanced `for` loop (`for(:)`), the `while` loop, and the `do-while` loop
- Understanding control transfer with the `yield`, `break`, `continue`, and `return` statements, including labeled statements

Java SE 17 Developer Exam O	
[4.1] Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements	§4.1, p. 152 to §4.13, p. 184
Java SE 11 Developer Exam O	
[2.1] Create and use loops, if/else, and switch statements	§4.1, p. 152 to §4.8, p. 176

Control flow statements determine *the flow of control* in a program during execution, meaning the order in which statements are executed in a running program. There are three main categories of control flow statements:

- *Selection* statements: `if`, `if-else`, and `switch`
- *Iteration* statements: `while`, `do-while`, basic `for`, and enhanced `for` loops
- *Transfer* statements: `yield`, `break`, `continue`, and `return`

Each category of statements is discussed in subsequent sections.

4.1 Selection Statements

Java provides selection statements that allow the program to choose between alternative actions during execution. The choice is based on criteria specified in the selection statement. These selection statements are

- The simple `if` statement
- The `if-else` statement
- The `switch` statement and the `switch` expression

The Simple `if` Statement

The simple `if` statement has the following syntax:

```
if (condition)
  statement
```

It is used to decide whether an action is to be performed or not, based on a *condition*. The action to be performed is specified by *statement*, which can be a single statement or a code block. The *condition* must evaluate to a `boolean` or `Boolean` value. In the latter case, the `Boolean` value is unboxed to the corresponding `boolean` value.

The semantics of the simple `if` statement are straightforward. The *condition* is evaluated first. If its value is `true`, *statement* (called the `if` block) is executed and then execution continues with the rest of the program. If the value is `false`, the `if` block is skipped and execution continues with the rest of the program. The semantics are illustrated by the activity diagram in Figure 4.1a.

Figure 4.1 *Activity Diagram for* `if` *Statements*

In the following examples of the `if` statement, it is assumed that the variables and the methods have been appropriately defined:

Click here to view code image

```
if (emergency)             // emergency is a boolean variable
  operate();

if (temperature > critical)
  soundAlarm();

if (isLeapYear() && endOfCentury())
  celebrate();
```

```
if (catIsAway()) {          // Block
  getFishingRod();
  goFishing();
}
```

Note that *statement* can be a *block*, and the block notation is necessary if more than one statement is to be executed when the *condition* is `true`.

Since the *condition* evaluates to a `boolean` value, it avoids a common programming error: using an expression of the form `(a=b)` as the condition, where inadvertently an assignment operator is used instead of a relational operator. The compiler will flag this as an error, unless both `a` and `b` are boolean.

Note that the `if` block can be any valid statement. In particular, it can be the empty statement (`;`) or the empty block (`{}`). A common programming error is inadvertent use of the empty statement.

Click here to view code image

```
if (emergency); // Empty if block
  operate();    // Executed regardless of whether it was an emergency
```

The `if-else` Statement

The `if-else` statement is used to decide between two actions, based on a *condition*. It has the following syntax:

```
if (condition)
  statement1
else
  statement2
```

The *condition* is evaluated first. If its value is `true` (or unboxed to `true`), *statement*$_1$ (the `if` block) is executed and then execution continues with the rest of the program. If the value is `false` (or unboxed to `false`), *statement*$_2$ (the `else` block) is executed and then execution continues with the rest of the program. In other words, one of two mutually exclusive actions is performed. The `else` clause is optional; if omitted, the construct is equivalent to the simple `if` statement. The semantics are illustrated by the activity diagram in Figure 4.1b.

In the following examples of the `if-else` statement, it is assumed that all variables and methods have been appropriately defined:

Click here to view code image

```
if (emergency)
  operate();
else
  joinQueue();

if (temperature > critical)
  soundAlarm();
else
  businessAsUsual();

if (catIsAway()) {
  getFishingRod();
  goFishing();
} else
  playWithCat();
```

Since actions can be arbitrary statements, the `if` statements can be nested.

Click here to view code image

```
if (temperature >= upperLimit) {          // (1)
  if (danger)                             // (2) Simple if.
    soundAlarm();
  if (critical)                           // (3)
    evacuate();
  else                                    // Goes with if at (3).
    turnHeaterOff();
} else                                    // Goes with if at (1).
    turnHeaterOn();
```

The use of block notation, `{}`, can be critical to the execution of `if` statements. The `if` statements (A) and (B) in the following examples do *not* have the same meaning. The `if` statements (B) and (C) are the same, with extra indentation used in (C) to make the meaning evident. Leaving out the block notation in this case could have catastrophic consequences: The heater could be turned on when the temperature is above the upper limit.

Click here to view code image

```
// (A):
if (temperature > upperLimit) {          // (1) Block notation.
  if (danger) soundAlarm();              // (2)
} else                                   // Goes with if at (1).
  turnHeaterOn();

// (B):
if (temperature > upperLimit)            // (1) Without block notation.
  if (danger) soundAlarm();              // (2)
else turnHeaterOn();                     // Goes with if at (2).

// (C):
if (temperature > upperLimit)            // (1)

  if (danger)                            // (2)
    soundAlarm();
  else                                   // Goes with if at (2).
```

```
    turnHeaterOn();
```

The rule for matching an `else` clause is that an `else` clause always refers to the nearest `if` that is not already associated with another `else` clause. Block notation and proper indentation can be used to make the meaning obvious.

Cascading of `if-else` statements comprises a sequence of nested `if-else` statements where the `if` block of the next `if-else` statement is joined to the `else` clause of the previous `if-else` statement. The decision to execute a block is then based on all the conditions evaluated so far.

Click here to view code image

```
if (temperature >= upperLimit) {                          // (1)
  soundAlarm();
  turnHeaterOff();
} else if (temperature < lowerLimit) {                    // (2)
  soundAlarm();

  turnHeaterOn();
} else if (temperature == (upperLimit-lowerLimit)/2) {    // (3)
  doingFine();
} else                                                    // (4)
  noCauseToWorry();
```

The block corresponding to the first `if` condition that evaluates to `true` is executed, and the remaining `if` statements are skipped. In the preceding example, the block at (3) will execute only if the conditions at (1) and (2) are `false` and the condition at (3) is `true`. If none of the conditions is `true`, the block associated with the last `else` clause is executed. If there is no last `else` clause, no actions are performed.

4.2 The `switch` Statement

The `switch` construct implements a *multi-way branch* that allows program control to be transferred to a specific entry point in the code of the `switch` block based on a computed value. Java has two variants of the `switch` construct (the `switch` *statement* and the `switch` *expression*), and each of them can be written in two different ways (one using the *colon notation* and the other using the *arrow notation*). This section covers the two forms of the `switch` statement. Particular details of the `switch` expression are covered in the next section (p. 164).

The `switch` Statement with the Colon (`:`) Notation

We will first look at the `switch` *statement* defined using the *colon notation*, illustrated in Figure 4.2.

Figure 4.2 *Form of the* `switch` *Statement with the Colon Notation*

```
switch (selector_expression) {
 // Switch block with statement groups defined using colon notation:
 case CC:                                   statements
 case CC1: case CC2: ... case CCn:          statements
 case CC3, CC4, ..., CCm:                   statements
 ...
 default: ...
}
```

Conceptually, the `switch` statement can be used to choose one among many alternative actions, based on the value of an expression. The syntax of the `switch` statement comprises a *selector expression* followed by a `switch` *block*. The selector expression must evaluate to a value whose type must be one of the following:

- A primitive data type: `char`, `byte`, `short`, or `int`
- A wrapper type: `Character`, `Byte`, `Short`, or `Integer`
- An enum type (§5.13, p. 287)
- The type `String` (§8.4, p. 439)

Note that the type of the selector expression cannot be `boolean`, `long`, or floating-point. The statements in the `switch` block can have `case` *labels*, where each `case` label specifies one or more `case` *constants* (*CC*), thereby defining entry points in the `switch` block where control can be transferred depending on the value of the selector expression. The `switch` block must be compatible with the type of the selector expression, otherwise a compile-time error occurs.

The execution of the `switch` statement proceeds as follows:

- The selector expression is evaluated first. If the value is a wrapper type, an unboxing conversion is performed (§2.3, p. 45). If the selector expression evaluates to `null`, a `NullPointerException` is thrown.
- The value of the selector expression is compared with the constants in the `case` labels. Control is transferred to the start of the *statements* associated with the `case` label that

has a `case` constant whose value is equal to the value of the selector expression. Note that a colon (:) prefixes the associated statements that can be any *group of statements*, including a statement block. After execution of the associated statements, control *falls through* to the *next* group of statements, unless this was the last group of statements declared or control was transferred out of the `switch` statement.

- If no `case` label has a `case` constant that is equal to the value of the selector expression, the statements associated with the `default` label are executed. After execution of the associated statements, control *falls through* to the *next* group of statements, unless this was the last group of statements declared or control was transferred out of the `switch` statement.

Figure 4.3 illustrates the flow of control through a `switch` statement where the `default` label is declared last and control is not transferred out of the `switch` statement in the preceding group of statements.

Figure 4.3 *Activity Diagram for the* `switch` *Statement with the Colon Notation*

All `case` labels (including the `default` label) are optional and can be defined in any order in the `switch` block. All `case` labels and the `default` label are separated from their associated group of statements by a colon (`:`). A list of `case` labels can be associated with the same statements, and a `case` label can specify a comma-separated list of `case` constants. At most, one `default` label can be present in a `switch` statement. If no valid `case` labels are found and the `default` label is omitted, the whole `switch` statement is skipped.

The `case` *constants* (*CC*) in the `case` labels are constant expressions whose values must be unique, meaning no duplicate values are allowed. In fact, a `case` constant must be a compile-time constant expression whose value is *assignable* to the type of the selector expression (§2.4, p. 46). In particular, all `case` constant values must be in the range of the type of the selector expression. The type of a `case` constant cannot be `boolean`, `long`, or floating-point.

The compiler is able to generate efficient code for a `switch` statement, as this statement only tests for *equality* between the selector expression and the constant expressions of the `case` labels, so as to determine which code to execute at runtime. In contrast, a sequence of `if` statements determines the flow of control at runtime, based on arbitrary conditions which might be determinable only at runtime.

In Example 4.1, depending on the value of the `howMuchAdvice` parameter, different advice is printed in the `switch` statement at (1) in the method `dispenseAdvice()`. The example shows the output when the value of the `howMuchAdvice` parameter is `LOTS_OF_ADVICE`. In the `switch` statement, the associated statement at (2) is executed, giving one piece of advice. Control then falls through to the statement at (3), giving the second piece of advice. Control next falls through to (4), dispensing the third piece of advice, and finally execution of the `break` statement at (5) causes control to exit the `switch` statement. Without the `break` statement at (5), control would continue to fall through the remaining statements—in this case, to the statement at (6) being executed. Execution of the `break` statement in a `switch` block transfers control out of the `switch` statement (p. 180). If the parameter `howMuchAdvice` has the value `MORE_ADVICE`, then the advice at both (3) and (4) is given. The value `LITTLE_ADVICE` results in only one piece of advice at (4) being given. Any other value results in the `default` action, which announces that there is no advice.

The associated statement of a `case` label can be a *group* of statements (which need *not* be a statement block). The `case` label is prefixed to the first statement in each `case`. This is illustrated by the associated statements for the `case` constant `LITTLE_ADVICE` in Example 4.1, which comprises statements (4) and (5).

Example 4.1 *Fall-Through in a* `switch` *Statement with the Colon Notation*

Click here to view code image

```
public class Advice {

  private static final int LITTLE_ADVICE = 0;
  private static final int MORE_ADVICE = 1;
  private static final int LOTS_OF_ADVICE = 2;

  public static void main(String[] args) {
    dispenseAdvice(LOTS_OF_ADVICE);
  }

  public static void dispenseAdvice(int howMuchAdvice) {
    switch (howMuchAdvice) {                                    // (1)
      case LOTS_OF_ADVICE: System.out.println("See no evil.");   // (2)
      case MORE_ADVICE:    System.out.println("Speak no evil."); // (3)
```

```
          case LITTLE_ADVICE:   System.out.println("Hear no evil.");   // (4)
                                break;                                 // (5)
          default:              System.out.println("No advice.");     // (6)
        }
      }
    }
```

Output from the program:

```
See no evil.
Speak no evil.
Hear no evil.
```

Several `case` labels can prefix the same group of statements. This is the equivalent of specifying the same `case` constants in a single `case` label. The latter syntax is preferable as it is more concise than the former. Such `case` constants will result in the associated group of statements being executed. This behavior is illustrated in Example 4.2 for the `switch` statement at (1).

At (2) in Example 4.2, three `case` labels are defined that are associated with the same action. At (3), (4), and (5), a list of `case` constants is defined for some of the `case` labels. Note also the use of the `break` statement to stop fall-through in the `switch` block after the statements associated with a `case` label are executed.

The first statement in the `switch` block must always have a `case` or `default` label; otherwise, it will be unreachable. This statement will never be executed because control can never be transferred to it. The compiler will flag this case (no pun intended) as an error. An empty `switch` block is perfectly legal, but not of much use.

Since each group of statements associated with a `case` label can be any arbitrary statement, it can also be another `switch` statement. In other words, `switch` statements can be nested. Since a `switch` statement defines its own local block, the `case` labels in an inner block do not conflict with any `case` labels in an outer block. Labels can be redefined in nested blocks; in contrast, variables cannot be redeclared in nested blocks (§6.6, p. 354). In Example 4.2, an inner `switch` statement is defined at (6), which allows further refinement of the action to take on the value of the selector expression in cases where multiple `case` labels are used in the outer `switch` statement. A `break` statement terminates the innermost `switch` statement in which it is executed.

The print statement at (7) is always executed for the case constants `9`, `10`, and `11`.

Note that the `break` statement is the last statement in the group of statements associated with each `case` label. It is easy to think that the `break` statement is a part of the `switch` statement syntax, but technically it is not.

Example 4.2 *Nested* `switch` *Statements with the Colon Notation*

Click here to view code image

```
public class Seasons {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                     // (1) Outer
      case 12: case 1: case 2:                                // (2)
        System.out.println("Snow in the winter.");
        break;
      case 3, 4: case 5:                                      // (3)
        System.out.println("Green grass in the spring.");
        break;
      case 6, 7, 8:                                           // (4)
        System.out.println("Sunshine in the summer.");
        break;
      case 9, 10, 11:                                         // (5)
        switch(monthNumber) { // Nested switch                   (6) Inner
          case 10:
            System.out.println("Halloween.");
            break;
          case 11:
            System.out.println("Thanksgiving.");
            break;
        } // End nested switch
        // Always printed for case constant 9, 10, 11
        System.out.println("Yellow leaves in the fall.");     // (7)
        break;
      default:
        System.out.println(monthNumber + " is not a valid month.");
    }
  }
}
```

Output from the program:

Click here to view code image

```
Thanksgiving.
Yellow leaves in the fall.
```

The `switch` Statement with the Arrow (->) Notation

The form of the `switch` statement with the arrow notation is shown in Figure 4.4. This form defines *switch rules* in which each `case` label is associated with a corresponding action using the arrow (->) notation.

Figure 4.4 *Form of the* `switch` *Statement with the Arrow Notation*

```
switch (selector_expression) {
  // Switch block with switch rules defined using arrow notation:
  case CC                      -> expression_statement;
  case CC1, CC2, ..., CCm      -> block
  case CC4                     -> throw_statement
  ...
  default                      -> ...
}
```

Compared to the `switch` statement with the colon notation (Figure 4.2), there are a few things to note.

First, although the `case` labels (and the `default` label) are specified similarly, the arrow notation does not allow multiple `case` labels to be associated with a common action. However, the same result can be achieved by specifying a single `case` label with a list of `case` constants, thereby associating the `case` constants with a common action.

Second, the action that can be associated with the `case` labels in switch rules is restricted. The `switch` statement with the colon notation allows a group of statements, but the `switch` statement with the arrow notation only allows the following actions to be associated with `case` labels:

- *An expression statement (§3.3, p. 101)*

  By far, the canonical action of a `case` label in a switch rule is an expression statement. Such an expression statement is always terminated by a semicolon (`;`). Typically, the value returned by the expression statement is discarded. In the examples below, what is important is the side effect of evaluating the expression statements.

  Click here to view code image

  ```
  ...
  case PASSED -> ++numbersPassed;
  case FAILED -> ++numbersFailed;
  ...
  ```

- *A block (§6.6, p. 354)*

  A block of statements can be used if program logic should be refined.

  Click here to view code image

  ```
  ...
  case ALARM ->  { soundTheAlarm();
                   callTheFireDepartment(); }
  ...
  ```

- *Throw an exception (§7.4, p. 386)*

  The switch rule below throws an exception when the value of the selector expression does not match any `case` constants:

  Click here to view code image

  ```
  ...
  default -> throw new IllegalArgumentException("Not a valid value");
  ...
  ```

Third, the execution of the switch rules is *mutually exclusive* (Figure 4.5). Once the action in the switch rule has completed execution, the execution of the `switch` statement terminates. This is illustrated in Figure 4.5 where only one expression statement is executed, after which the `switch` statement also terminates. There is no fall-through and the `break` statement is not necessary.

Figure 4.5 *Activity Diagram for the* `switch` *Statement with the Arrow Notation*

Example 4.3 is a refactoring of Example 4.2 with a `switch` statement with the arrow notation. At (2), (3), (4), and (8), the action executed is an expression statement, whereas at (5), the action executed is a block. Using switch rules results in compact and elegant code that also improves the readability of the `switch` statement.

Example 4.3 *Nested* `switch` *Statements with the Arrow Notation*

Click here to view code image

```
public class SeasonsII {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                             // (1) O
      case 12, 1,  2 -> System.out.println("Snow in the winter.");    // (2)
      case 3,  4,  5 -> System.out.println("Green grass in the spring.");    //
      case 6,  7,  8 -> System.out.println("Sunshine in the summer.");       //
      case 9, 10, 11 -> {                                             // (5)
        switch(monthNumber) { // Nested switch                            (6) I
          case 10 -> System.out.println("Halloween.");
          case 11 -> System.out.println("Thanksgiving.");
        }
        // Always printed for case constants 9, 10, 11:
        System.out.println("Yellow leaves in the fall.");             // (7)
      }
      default -> throw new IllegalArgumentException(monthNumber +
                                             " is not a valid month.");// (8)
    }
  }
```

```
}
```

Output from the program:

Click here to view code image

```
Thanksgiving.
Yellow leaves in the fall.
```

Using Strings as `case` *Constants*

Example 4.4 illustrates using strings in a `switch` statement. The thing to note is what constitutes a constant string expression that can be used as a `case` constant. The `case` constants at (3), (4), (5), and (6) are all valid *constant string expressions*, as the compiler can figure out their value at compile time. String literals, used at (3) and (6), and constant field values, declared at (1) and (2a) and used at (4) and (5), are all valid `case` constants. In contrast, the `HOT` reference from declarations (2b) and (2c) cannot be used as a `case` constant. From the declaration at (2a), the compiler cannot guarantee that the value of the reference will not change at runtime. From the declaration at (2c), it cannot deduce the value at compile time, as the constructor must be run to construct the value.

Switching on strings is essentially based on equality comparison of integer values that are hash values of strings, followed by an object equality test to rule out the possibility of collision between two different strings having the same hash value. Switching on strings should be used judiciously, as it is less efficient than switching on integers. Switching on strings is not advisable if the values being switched on are not already strings.

Example 4.4 *Strings in a* `switch` *Statement*

Click here to view code image

```
public class SwitchingOnAString {
  public static final String MEDIUM = "Medium";       // (1)
  public static final String HOT = "Hot";             // (2a)
//public static       String HOT = "Hot";             // (2b) Not OK as case l
//public static final String HOT = new String("Hot"); // (2c) Not OK as case l
  public static void main(String[] args) {
    String spiceLevel = "Medium_Hot";
    switch (spiceLevel) {
      case "Mild",
            MEDIUM + "_" + HOT -> System.out.println("Enjoy your meal!"); // (
      case HOT                -> System.out.println("Have fun!");        // (
      case "Suicide"          -> System.out.println("Good luck!");       // (
      default                 -> System.out.println("You being funny?");
    }
  }
}
```

Output from the program:

```
Enjoy your meal!
```

Using Enum Constants as `case` *Constants*

Example 4.5 illustrates the use of enum types (§5.13, p. 287) in a `switch` statement with the arrow notation. The enum type `SpiceGrade` is defined at (1). The type of the selector expression at (2) is the enum type `SpiceGrade`. Note that the enum constants are *not* specified with their fully qualified name (see (3a)). Using the fully qualified name results in a compile-time error, as shown at (3b). Only enum constants that have the same enum type as the selector expression can be specified as `case` label values.

The semantics of the `switch` statement are the same as described earlier. Switching on enum values is essentially based on equality comparison of unique integer values that are ordinal values assigned by the compiler to the constants of an enum type.

When the switch rules cover *all* values of the selector expression type, the `switch` statement is said to be *exhaustive*. Non-exhaustive `switch` statements are a common cause of programming errors. It is up to the programmer to ensure that the `switch` statement is exhaustive, as the compiler does not provide any help in this regard for the `switch` statement. Judicious use of the `default` label should be considered, as illustrated in the examples provided in this section that use the `switch` statement.

Example 4.5 *Enums in a* `switch` *Statement*

Click here to view code image

```
enum SpiceGrade { MILD, MEDIUM, MEDIUM_HOT, HOT, SUICIDE; }   // (1)

public class SwitchingFun {
  public static void main(String[] args) {
    SpiceGrade spicing = SpiceGrade.HOT;
    switch (spicing) {                                    // (2)
      case HOT ->  System.out.println("Have fun!");    // (3a) OK!
//    case SpiceGrade.HOT                                 // (3b) Compile-time err
//           -> System.out.println("Have fun!");
      case SUICIDE -> System.out.println("Good luck!");
      default -> System.out.println("Enjoy your meal!");
    }
  }
}
```

Output from the program:

```
Have fun!
```

4.3 The `switch` Expression

A `switch` expression evaluates to a value, as opposed to a `switch` statement that does not. Conceptually we can think of a `switch` expression as an augmented `switch` statement that returns a value. We look at both forms of the `switch` expression, defined using the colon notation and the arrow notation, how it yields a value, and compare it to the `switch` statement. The `switch` expression is analogous to the `switch` statement, except for the provision to return a value.

The `yield` Statement

The `yield` statement in a `switch` expression is analogous to the `break` statement in a `switch` statement. It can only be used in a `switch` expression, where the identifier `yield` is a *contextual keyword* only having a special meaning in the context of a `switch` expression.

```
yield expression;
```

Execution of the `yield` statement results in the expression being evaluated, and its value being returned as the value of the `switch` expression.

The `switch` Expression with the Colon (`:`) Notation

The `switch` expression with the colon notation has the same form as the `switch` statement with the colon notation (Figure 4.2), except that the execution of the `switch` body results in a value (or it throws an exception).

Example 4.6 is a reworking of Example 4.2 with seasons, where the group of statements associated with a `case` label print information about the season and return a constant of the enum type `Season` that is defined at (1). Note that the `yield` statement is the last statement in the group of statements associated with each `case` label. Execution of the `yield` statement results in its expression being evaluated, and its value being returned as the value of the `switch` expression, thereby also terminating the execution of the `switch` expression. Not surprisingly, a `break` or a `return` statement is not allowed in a `switch` expression. Note that the `switch` expression is on the right-hand side of the assignment statement defined at (2) and is terminated by a semicolon (`;`).

The *fall-through* of execution in the `switch` expression with the colon notation is analogous to that of the `switch` statement with the colon notation (Figure 4.3). If a group of statements associated with a `case` label does not end in a `yield` statement, execution continues with the next group of statements, if any.

The `switch` expression with the colon notation must be *exhaustive*, meaning the `case` labels, and if necessary the `default` label, must cover *all* values of the selector expression type. Non-exhaustive `switch` expressions will result in a compile-time error. The `default` label is typically used to make the `switch` expression exhaustive. In Example 4.6, the type of the selector expression is `int`, but the `case` labels only cover the `int` values from 1 to 12. A `default` label is necessary to cover the other `int` values or to throw an exception, as in this case, and make the `switch` expression exhaustive.

Example 4.6 *A* `yield` *Statement in a* `switch` *Expression with the Colon Notation*

Click here to view code image

```
public class SeasonsIII {

  enum Season { WINTER, SPRING, SUMMER, FALL }                // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                     // (2)
      case 12: case 1: case 2:                                // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                  // (4)
      case 3, 4: case 5:                                      // (5)
        System.out.println("Green grass in the spring.");
        yield Season.SPRING;                                  // (6)
      case 6, 7, 8:                                           // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                                  // (8)
      case 9, 10, 11:                                         // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                    // (10)
      default:                                                // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month."
    };                                                        // (12)
    System.out.println(season);
  }
}
```

Output from the program:

Click here to view code image

```
Yellow leaves in the fall.
FALL
```

The `switch` Expression with the Arrow (->) Notation

The `switch` expression with the arrow notation also has the same form as the `switch` statement with the arrow notation (Figure 4.4), except that the execution of the `switch` body must result in a value (or it must throw an exception).

The execution of the switch rules in a `switch` expression is *mutually exclusive*, analogous to the switch rules in a `switch` statement (Figure 4.5). Once the action in the switch rule has completed execution, the value computed by the action is returned and the execution of the `switch` expression terminates. There is no fall-through and no `break` statement is allowed.

Whereas the actions in the switch rules of a `switch` statement only allowed an *expression statement* (Figure 4.5), the actions in the switch rules of a `switch` expression allow *any expression*, in addition to allowing a block or throwing an exception, as in the switch rules of a `switch` statement.

- *Any expression (Chapter 2, p. 29)*

  By far, the canonical action of a `case` label in a switch rule of a `switch` expression is an arbitrary expression. Such an expression is always terminated by a semicolon (`;`). The expression value is returned as the value of the `switch` expression whose execution is then terminated. Note that no `yield` statement is necessary or allowed.

  Click here to view code image

  ```
  ...
  case 1 -> "ONE";
  case 2 -> yield "two";        // Compile-time error!
  ...
  ```

- *A block (§6.6, p. 354)*

  A *block* of statements can be used if program logic should be refined, but the last statement in the block should be a `yield` statement to return its value and terminate the execution of the `switch` expression (alternatively, the last statement can be a `throw` statement). In a `switch` expression with the arrow notation, the `yield` statement is only allowed as the last statement in a block that constitutes the action in a switch rule.

Click here to view code image

```
...
case ALARM     -> { soundTheAlarm();
                    callTheFireDepartment();
                    yield Status.EVACUATE; }  // OK
case ALL_CLEAR -> { yield Status.NORMAL; // Compile-time error: not la
                    standDown(); }       //                     in the
...
```

- *Throw an exception (§7.4, p. 386)*

  As the switch rules must be *exhaustive*, one way to achieve exhaustiveness is to throw an exception as the action in the `default` label.

Click here to view code image

```
...
default -> throw new IllegalArgumentException("Not a valid value");
...
```

Example 4.6 has been refactored to use the `switch` expression with the arrow notation in Example 4.7. Each action associated with a `case` label of a switch rule is a block of statements, where a `yield` statement is the last statement in a block. If this is not the case, the code will not compile.

The `switch` expression with the arrow notation must also be *exhaustive*. Again a non-exhaustive `switch` expression with the arrow notation will result in a compile-time error. In Example 4.7, the type of the selector expression is `int`, but the switch rules only cover the `int` values from 1 to 12. A `default` label is necessary to make the `switch` expression exhaustive, as shown at (11).

Example 4.7 *Statement Blocks in a* `switch` *Expression with the Arrow Notation*

Click here to view code image

```
public class SeasonsIV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                    // (1)
  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                     // (2)
      case 12, 1, 2 -> {                                      // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                  // (4)
      }
      case 3, 4, 5 -> {                                       // (5)
        System.out.println("Green grass in the spring.");
        yield Season.SPRING;                                  // (6)
      }
      case 6, 7, 8 -> {                                       // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                                  // (8)
      }
      case 9, 10, 11 -> {                                     // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                    // (10)
      }
      default ->                                              // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month."
    };                                                        // (12)
    System.out.println(season);
  }
}
```

Output from the program:

Click here to view code image

```
Yellow leaves in the fall.
FALL
```

Example 4.8 is a reworking of Example 4.7 that defines *expressions* as the actions in the switch rules. No `yield` statement is necessary or allowed in this case. The `switch` expression is also exhaustive.

Example 4.8 *Expression Actions in a* `switch` *Expression with the Arrow Notation*

Click here to view code image

```
public class SeasonsV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                      // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                           // (2)
      case 12,  1,  2 -> Season.WINTER;                             // (3)
      case  3,  4,  5 -> Season.SPRING;                             // (4)
      case  6,  7,  8 -> Season.SUMMER;                             // (5)
      case  9, 10, 11 -> Season.FALL;                               // (6)
      default         -> throw new IllegalArgumentException(monthNumber +
                                                           " not a valid month
    };
    System.out.println(season);
  }
}
```

Output from the program:

```
FALL
```

The `switch` expression can only evaluate to a single value. Multiple values can be returned by constructing an object with the required values and returning the object as a result of evaluating the `switch` expression. Record classes are particularly suited for this purpose (§5.14, p. 299). The `switch` expression at (3) in Example 4.9 returns an object of the record class `SeasonInfo`, defined at (2), to return the month number and the season in which it occurs.

Example 4.9 *Returning Multiple Values as a Record from a* `switch` *Expression*

Click here to view code image

```
public class SeasonsVI {

  enum Season { WINTER, SPRING, SUMMER, FALL }                              //
  record SeasonInfo(int month, Season season) {}                            //

  public static void main(String[] args) {
    int monthNumber = 11;
    SeasonInfo seasonInfo = switch(monthNumber) {                           //
      case 12,  1,  2 -> new SeasonInfo(monthNumber, Season.WINTER);        //
      case  3,  4,  5 -> new SeasonInfo(monthNumber, Season.SPRING);        //
      case  6,  7,  8 -> new SeasonInfo(monthNumber, Season.SUMMER);        //
      case  9, 10, 11 -> new SeasonInfo(monthNumber, Season.FALL);          //
      default         -> throw new IllegalArgumentException(monthNumber +
                                                           " not a valid month
    };
    System.out.println(seasonInfo);
  }
}
```

Output from the program:

Click here to view code image

```
SeasonInfo[month=11, season=FALL]
```

Local Variable Scope in the `switch` Body

The *scope* of a local variable declared in a `switch` statement or a `switch` expression is the *entire* `switch` block. Any local block in the `switch` body introduces a new *local scope*. Any local variable declared in it has *block scope*, and therefore, is only accessible in that block. A local variable declared in an enclosing local scope cannot be redeclared in a nested local scope (§6.6, p. 354).

Summary of the `switch` Statement and the `switch` Expression

Table 4.1 summarizes the features of the `switch` statement and the `switch` expression, and provides a comparison of the two constructs.

Table 4.1 *Comparing the* `switch` *Statement and the* `switch` *Expression*

Notation	The `switch` statement	The `switch` expression
The colon (:) notation: `case` *label*: *statements*	Executes statements associated with the matching `case` label. • Fall-through can occur. • No compile-time check for exhaustiveness. • Only `break` and `return` statements allowed to control fall-through.	Executes statements associated with the matching `case` label, but must have a `yield` statement to return a value. • Fall-through can occur. • Compile-time check for exhaustiveness. • Only `yield` statement allowed to control fall-through.
The arrow (->) notation: `case` *label* -> *expression*;	Action associated with a switch rule can be an *expression statement*, can be a *block*, or can *throw an exception*. • Mutually exclusive switch rules: no fall-through can occur. • No compile-time check for exhaustiveness. • `break` and `return` statements allowed.	Action associated with a switch rule can be an *expression*, can be a *block*, or can *throw an exception*. • Mutually exclusive switch rules: no fall-through can occur. • Compile-time check for exhaustiveness. • No `break` or `return` statements allowed. • Must return a value that is either the value of a stand-alone *expression* or the value of the expression in a `yield` statement that can occur as the last statement in a *block*.

Review Questions

4.1 What will be the result of attempting to compile and run the following class?

Click here to view code image

```
public class IfTest {
  public static void main(String[] args) {
    if (true)
    if (false)
    System.out.println("a");
```

```
      else
      System.out.println("b");
    }
  }
```

Select the one correct answer.

a. The code will fail to compile because the syntax of the `if` statement is incorrect.

b. The code will fail to compile because the compiler will not be able to determine which `if` statement the `else` clause belongs to.

c. The code will compile correctly and will display the letter `a` at runtime.

d. The code will compile correctly and will display the letter `b` at runtime.

e. The code will compile correctly but will not display any output.

4.2 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Switching {
  public static void main(String[] args) {
    final int iLoc = 3;
    switch (6) {
      case 1:
      case iLoc:
      case 2 * iLoc:
        System.out.println("I am not OK.");
      default:
        System.out.println("You are OK.");
      case 4:
        System.out.println("It's OK.");
    }
  }
}
```

Select the one correct answer.

a. The code will fail to compile because of the `case` label value `2 * iLoc`.

b. The code will fail to compile because the `default` label is not specified last in the `switch` statement.

c. The code will compile correctly and will print the following at runtime:

```
I am not OK.
You are OK.
It's OK.
```

d. The code will compile correctly and will print the following at runtime:

```
You are OK.
It's OK.
```

e. The code will compile correctly and will print the following at runtime:

```
It's OK.
```

4.3 Which code option will print the string `"Prime"`?

Select the one correct answer.

a. Click here to view code image

```
char value = 3;
String result = "Unknown";
switch (value) {
  case 2,3,5,7:   result = "Prime";
  case 1,4,6,8,9: result = "Composite";
}
System.out.println(result);
```

b. Click here to view code image

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
};
System.out.println(result);
```

c. Click here to view code image

```
char value = 3;
String yield =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
  default:        yield "Unknown";
};
System.out.println(yield);
```

d. Click here to view code image

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7   -> "Prime";
  case 1,4,6,8,9 -> "Composite";
  default: { yield "Unknown"; }
};
System.out.println(result);
```

4.4 Given the following code:

Click here to view code image

```
public class RQ462 {
  public static void main(String[] args) {
    int price = 1;
    int discount = switch (price) {
      case 5, 1, 2 -> price - 1;
      case 4, 3, 6 -> price - 2;
      default      -> 0;
    };
    System.out.println(discount);
```

```
        }
    }
```

What is the result?

Select the one correct answer.

a. `0`
b. `1`
c. `-1`
d. `-2`
e. The program will throw an exception at runtime.
f. The program will fail to compile.

4.4 Iteration Statements

Loops allow a single statement or a statement block to be executed repeatedly (i.e., iterated). A boolean condition (called the *loop condition*) is commonly used to determine when to terminate the loop. The statements executed in the loop constitute the *loop body*.

Java provides four language constructs for loop construction:

- The `while` statement
- The `do-while` statement
- The *basic* `for` statement
- The *enhanced* `for` statement

These loops differ in the order in which they execute the loop body and test the loop condition. The `while` loop and the basic `for` loop test the loop condition *before* executing the loop body, whereas the `do-while` loop tests the loop condition *after* execution of the loop body.

The *enhanced* `for` loop (also called the *for-each* loop) simplifies iterating over arrays and collections. We will use the notations `for(;;)` and `for(:)` to designate the basic `for` loop and the enhanced `for` loop, respectively.

4.5 The `while` Statement

The syntax of the `while` loop is

```
while (loop_condition)
  loop_body
```

The *loop condition* is evaluated before executing the *loop body*. The `while` statement executes the *loop body* as long as the *loop condition* is `true`. When the *loop condition* becomes `false`, the loop is terminated and execution continues with any statement immediately following the loop. If the *loop condition* is `false` to begin with, the *loop body* is not executed at all. In other words, a `while` loop can execute zero or more times. The *loop condition* must evaluate to a `boolean` or a `Boolean` value. In the latter case, the reference value is unboxed to a `boolean` value. The flow of control in a `while` statement is shown in Figure 4.6.

Figure 4.6 *Activity Diagram for the* `while` *Statement*

The `while` statement is normally used when the number of iterations is not known.

```
while (noSignOfLife())
  keepLooking();
```

Since the *loop body* can be any valid statement, inadvertently terminating each line with the empty statement (`;`) can give unintended results. Always using a block statement as the *loop body* helps to avoid such problems.

Click here to view code image

```
while (noSignOfLife());     // Empty statement as loop body!
  keepLooking();            // Statement not in the loop body.
```

4.6 The `do-while` Statement

The syntax of the `do-while` loop is

```
do
loop_body
while (loop_condition);
```

In a `do-while` statement, the *loop condition* is evaluated *after* executing the *loop body*. The *loop condition* must evaluate to a `boolean` or `Boolean` value. The value of the *loop condition* is subjected to unboxing if it is of the type `Boolean`. The `do-while` statement executes the *loop body* until the *loop condition* becomes `false`. When the *loop condition* becomes `false`, the loop is terminated and execution continues with any statement immediately following the loop. Note that the *loop body* is executed at least once. Figure 4.7 illustrates the flow of control in a `do-while` statement.

Figure 4.7 *Activity Diagram for the* `do-while` *Statement*

The *loop body* in a `do-while` loop is invariably a statement block. It is instructive to compare the `while` and `do-while` loops. In the examples that follow, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

Click here to view code image

```
while (cat.isAway()) {        // (1)
  mice.play();
}
do {                          // (2)
  mice.play();
} while (cat.isAway());
```

4.7 The `for(;;)` Statement

The `for(;;)` loop is the most general of all the loops. It is mostly used for *counter-controlled loops*, in which the number of iterations is known beforehand.

The syntax of the loop is as follows:

Click here to view code image

```
for (initialization; loop_condition; update_expression)
     loop_body
```

The *initialization* usually declares and initializes a *loop variable* that controls the execution of the *loop body*. The loop body can be a single statement or a statement block. The *loop condition* must evaluate to a `boolean` or `Boolean` value. In the latter case, the reference value is converted to a `boolean` value by unboxing. The *loop condition* usually involves the loop variable, and if the loop condition is `true`, the *loop body* is executed; otherwise, execution continues with any statement following the `for(;;)` loop. After each iteration (i.e., execution of the loop body), the *update expression* is executed. This usually modifies the value of the loop variable to ensure eventual loop termination. The *loop condition* is then tested to determine whether the loop body should be executed again. Note that the *initialization* is executed only once, on entry into the loop. The semantics of the `for(;;)` loop are illustrated in Figure 4.8, and are summarized by the following equivalent `while` loop code template:

Click here to view code image

```
initialization
while (loop_condition) {
   loop_body
   update_expression
}
```

Figure 4.8 *Activity Diagram for the* `for` *Statement*

The following code creates an `int` array and sums the values in the array:

Click here to view code image

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++)   // (1)
  sum += array[index];
```

The loop variable `index` is declared and initialized in the *initialization* section of the loop. It is incremented in the *update expression* section. This loop is an example of a *forward* `for(;;)` loop, where the loop variable is incremented.

The next code snippet is an example of a *backward* `for(;;)` loop, where the loop variable is decremented to sum the values in the array:

Click here to view code image

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = array.length - 1; index >= 0; index--)
  sum += array[index];
```

It is instructive to compare the specification of the loop header in the forward and backward `for(;;)` loops in these examples.

The loop at (1) earlier showed how a declaration statement can be specified in the *initialization* section. Such a declaration statement can also specify a comma-separated list of variables:

Click here to view code image

```
for (int i = 0, j = 1, k = 2; ... ; ...) ...;       // (2)
```

The variables `i`, `j`, and `k` in the declaration statement all have type `int`. All variables declared in the *initialization* section are local variables in the `for(;;)` statement and obey the scope rules for local blocks, as do any variables declared in the *loop body*. The following code will not compile, however, as variable declarations of different types (in this case, `int` and `String`) require declaration statements that are terminated by semicolons:

Click here to view code image

```
for (int i = 0, String str = "@"; ... ; ...) ...;  // (3) Compile-time error
```

The *initialization* section can also be a comma-separated list of *expression* statements (§3.3, p. 101). Any value returned by an expression statement is discarded. For example, the loop at (2) can be rewritten by factoring out the variable declarations:

Click here to view code image

```
int i, j, k;                                        // Variable declaration
for (i = 0, j = 1, k = 2; ... ; ...) ...;           // (4) Only initialization
```

The *initialization* section is now a comma-separated list of three expressions. The expressions in such a list are always evaluated from left to right, and their values are discarded. Note that the variables `i`, `j`, and `k` at (4) are not local to the loop.

Declaration statements cannot be mixed with expression statements in the *initialization* section, as is the case at (5) in the following example. Factoring out the variable declaration, as at (6), leaves a legal comma-separated list of expression statements.

Click here to view code image

```
// (5) Not legal and ugly:
for (int i = 0, System.out.println("This won't do!"); flag; i++) {  // Error!
  // loop body
}

// (6) Legal, but still ugly:
int i;                                        // Declaration factored out.
for (i = 0, System.out.println("This is legal!"); flag; i++) {      // OK.
  // loop body
}
```

The *update expression* can also be a comma-separated list of expression statements. The following code specifies a `for(;;)` loop that has a comma-separated list of three variables in the *initialization* section, and a comma-separated list of two expressions in the *update expression* section:

Click here to view code image

```
// Legal usage but not recommended, as it can affect code comprehension.
int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };
for (int i = 0, j = sqMatrix[0].length - 1, asymDiagonal = 0;   // initializati
     i < sqMatrix.length;                                       // loop conditi
     i++, j--)                                           // update expression
  asymDiagonal += sqMatrix[i][j];                        // loop body
```

All sections in the `for(;;)` header are optional. Any or all of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the *loop condition* signifies that the loop condition is `true`. The “crab”, `(;;)`, can be used to construct an infinite loop, where termination is presumably achieved through code in the loop body (see the next section on transfer statements):

Click here to view code image

```
for (;;) doProgramming();        // Infinite loop
```

4.8 The `for(:)` Statement

The enhanced `for` loop is convenient when we need to iterate over an array or a collection, especially when some operation needs to be performed on each element of the array or collection. In this section we discuss iterating over arrays. In §15.2, p. 795, we take a look at the `for(:)` loop for iterating over collections.

Earlier in this chapter we used a `for(;;)` loop to sum the values of elements in an `int` array:

Click here to view code image

```
int sum = 0;
int[] intArray = {12, 23, 5, 7, 19};
for (int index = 0; index < intArray.length; index++) { // (1) using for(;;) l
  sum += intArray[index];
}
```

The `for(;;)` loop at (1) is rewritten using the `for(:)` loop in Figure 4.9.

Figure 4.9 *Enhanced* `for` *Statement*

The body of the loop is executed for each element in the array, where the variable `element` successively denotes the current element in the array `intArray`. When the loop terminates, the variable `sum` will contain the sum of all elements in the array. We do not care about the *position* of the elements in the array, just that the loop iterates over *all* elements of the array.

From Figure 4.9 we see that the `for(:)` loop header has two parts. The *expression* must evaluate to a reference value that refers to an *array* (or an object which implements the `Iterable<E>` interface (§15.2, p. 791)). The array can be an array of primitive values or objects, or even an array of arrays. The *expression* is evaluated only once. The *element declaration* specifies a local variable that can be assigned a value of the element type of the array. The type of the array `intArray` in Figure 4.9 is `int[]`, and the element type is `int`. The element variable of type `int` can be assigned `int` values from the array of `int`. However, this assignment might require either a boxing or an unboxing conversion, with optional widening conversion.

The element variable is local to the loop block and is not accessible after the loop terminates. Also, changing the value of the current variable does *not* change any value in the array. The *loop body*, which can be a simple statement or a statement block, is executed for each element in the array and there is no danger of any out-of-bounds errors.

The `for(:)` loop has its limitations. Specifically, we cannot change element values, and this kind of loop does not provide any provision for positional access using an index. The `for(:)` loop only increments by one and always in a forward direction. It does not allow iterations over several arrays simultaneously. Under such circumstances, the `for(;;)` loop can be more convenient.

Here are some code examples of legal `for(:)` loops:

Click here to view code image

```
// Some one-dim arrays:
int[]     intArray =    {10, 20, 30};
Integer[] intObjArray = {10, 20, 30};
String[]  strArray =    {"one", "two"};

// Some two-dim arrays:
Object[][] objArrayOfArrays = {intObjArray, strArray};
Number[][] numArrayOfArrays = {{1.5, 2.5}, intObjArray, {100L, 200L}};
int[][]    intArrayOfArrays = {{20}, intArray, {40}};

// Iterate over a String array.
// Expression type is String[], and element type is String.
// String is assignable to Object (widening conversion).
for (Object obj : strArray) {}

// Iterate over an int array.
// Expression type is int[], and element type is int.
// int is assignable to Integer (boxing conversion)
for (Integer iRef : intArrayOfArrays[0]){}

// Iterate over an Integer array.
// Expression type is Integer[], and element type is Integer.
// Integer is assignable to int (unboxing conversion)
for (int i : intObjArray){}

// Iterate over a two-dim int array.
// Outer loop: expression type is int[][], and element type is int[].
// Inner loop: expression type is int[], and element type is int.
for (int[] row : intArrayOfArrays)
  for (int val : row) {}
```

```
// Iterate over a two-dim Number array.
// Outer loop: expression type is Number[][], and element type is Number[].
// Outer loop: Number[] is assignable to Object[] (widening conversion).
// Inner loop: expression type is Object[], and element type is Object.
for (Object[] row : numArrayOfArrays)
  for (Object obj : row) {}

// Outer loop: expression type is Integer[][], and element type is Integer[].
// Outer loop: Integer[] is assignable to Number[].
// Inner loop: expression type is int[], and element type is int.
// Inner loop: int is assignable to double.
for (Number[] row : new Integer[][] {intObjArray, intObjArray, intObjArray})
  for (double num : intArray) {}
```

Here are some code examples of `for(:)` loops that are not legal:

Click here to view code image

```
// Expression type is Number[][], and element type is Number[].
// Number[] is not assignable to Number.
for (Number num : numArrayOfArrays) {}        // Compile-time error!

// Expression type is Number[], and element type is Number.
// Number is not assignable to int.
for (int row : numArrayOfArrays[0]) {}        // Compile-time error!

// Outer loop: expression type is int[][], and element type is int[].
// int[] is not assignable to Integer[].
for (Integer[] row : intArrayOfArrays)        // Compile-time error!
  for (int val : row) {}

// Expression type is Object[][], and element type is Object[].
// Object[] is not assignable to Integer[].
for (Integer[] row : objArrayOfArrays) {}     // Compile-time error!

// Outer loop: expression type is String[], and element type is String.
// Inner loop: expression type is String, which is not legal here. Not an arra
for (String str : strArray)
  for (char val : str) {}                     // Compile-time error!
```

When using the `for(:)` loop to iterate over an array, the two main causes of errors are an expression in the loop header that does not represent an array and/or an element type of the array that is not assignable to the local variable declared in the loop header.

4.9 Transfer Statements

Java provides the following language constructs for transferring control in a program:

- The `yield` statement (p. 164)
- The `break` statement
- The `continue` statement

- The `return` statement

The `throw` statement can also transfer control in a program (§7.4, p. 386).

4.10 Labeled Statements

A statement may have a *label*:

label **:** *statement*

A label is any valid identifier; it always immediately precedes the statement. Label names exist in their own namespace, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables. The scope of a label is the statement prefixed by the label, meaning that it cannot be redeclared as a label inside the labeled statement—analogous to the scope of local variables.

Click here to view code image

```
L1: if (i > 0) {
  L1: System.out.println(i);     // (1) Not OK. Label L1 redeclared.
}

L1: while (i < 0) {              // (2) OK.
  L2: System.out.println(i);
}

L1: {                            // (3) OK. Labeled block.
  int j = 10;
  System.out.println(j);
}

L1: try {                        // (4) OK. Labeled try-catch-finally block.
  int j = 10, k = 0;
  L2: System.out.println(j/k);
} catch (ArithmeticException ae) {
  L3: ae.printStackTrace();
} finally {

  L4: System.out.println("Finally done.");
}
```

A statement can have multiple labels:

Click here to view code image

```
LabelA: LabelB: System.out.println("Multiple labels. Use judiciously.");
```

A declaration statement cannot have a label:

Click here to view code image

```
L0: int i = 0;                   // Compile-time error!
```

A labeled statement is executed as if it were unlabeled, unless it is the `break` or `continue` statement. This behavior is discussed in the next two subsections.

4.11 The `break` Statement

The `break` statement comes in two forms: *unlabeled* and *labeled*.

Click here to view code image

```
break;             // the unlabeled form
break label;        // the labeled form
```

The unlabeled `break` statement terminates loops (`for(;;)`, `for(:)`, `while`, `do-while`) and `switch` statements, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, and execution continues after the enclosing statement.

In Example 4.10, the `break` statement at (1) is used to terminate a `for(;;)` loop. Control is transferred to (2) when the value of `i` is equal to `4` at (1), skipping the rest of the loop body and terminating the loop.

Example 4.10 also shows that the unlabeled `break` statement terminates only the innermost loop or `switch` statement that contains the `break` statement. The `break` statement at (3) terminates the inner `for(;;)` loop when `j` is equal to `2`, and execution continues in the outer `switch` statement at (4) after the `for(;;)` loop.

Example 4.10 *The* `break` *Statement*

Click here to view code image

```
class BreakOut {

  public static void main(String[] args) {
    System.out.println("i   sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4)
        break;                              // (1) Terminate loop. Control to
      // Rest of loop body skipped when i gets the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
    } // end for
    // (2) Continue here.
    int n = 2;
    switch (n) {
        case 1:
          System.out.println(n);
          break;
        case 2:
          System.out.println("Inner for(;;) loop: ");
          for (int j = 0; j <= n; j++) {
            if (j == 2)
              break;                          // (3) Terminate loop. Control to
            System.out.println("j = " + j);
          }
        default:
          System.out.println("default: n = " + n); // (4) Continue here.
      }
    }
  }
```

Output from the program:

Click here to view code image

```
i    sqrt(i)
1    1.00
2    1.41
3    1.73
Inner for(;;) loop:
j = 0
j = 1
default: n = 2
```

A labeled `break` statement can be used to terminate *any* labeled statement that contains the `break` statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

Click here to view code image

```
out:                            // Label.
{                               // (1) Labeled block.
  // ...
  if (j == 10) break out;       // (2) Terminate block. Control to (3).
  System.out.println(j);        // Rest of the block not executed if j == 10.
  // ...
}
// (3) Continue here.
```

In Example 4.11, the program continues to add the elements below the diagonal of a square matrix until the sum is greater than 10. Two nested `for` loops are defined at (1) and (2). The outer loop is labeled `outer` at (1). The unlabeled `break` statement at (3) transfers control to (5) when it is executed; that is, it terminates the inner loop and control is transferred to the statement after the inner loop. The labeled `break` statement at (4) transfers control to (6) when it is executed; that is, it terminates both the inner and outer loops, transferring control to the statement after the loop labeled `outer`.

Example 4.11 *Labeled* `break` *Statement*

Click here to view code image

```
class LabeledBreakOut {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) break;         // (3) Terminate inner loop. Control to (
          System.out.println("Element[" + i + ", " + j + "]: " +
                              squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) break outer; // (4) Terminate both loops. Control to (
        } // end inner loop
        // (5) Continue with update expression in the outer loop header.
      } // end outer loop
    // (6) Continue here.
    System.out.println("sum: " + sum);
  }
}
```

Output from the program:

```
Element[1, 0]: 2
Element[2, 0]: 9
sum: 11
```

4.12 The `continue` Statement

Like the `break` statement, the `continue` statement comes in two forms: *unlabeled* and *labeled*.

Click here to view code image

```
continue;             // the unlabeled form
continue label;        // the labeled form
```

The `continue` statement can be used only in a `for(;;)`, `for(:)`, `while`, or `do-while` loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible. In the case of the `while` and `do-while` loops, the rest of the loop body is skipped—that is, the current iteration is stopped, with execution continuing with the *loop condition*. In the case of the `for(;;)` loop, the rest of the loop body is skipped, with execution continuing with the *update expression*.

In Example 4.12, an unlabeled `continue` statement is used to skip an iteration in a `for(;;)` loop. Control is transferred to (2) when the value of `i` is equal to `4` at (1), skipping the rest of the loop body and continuing with the *update expression* in the `for(;;)` statement.

Example 4.12 *The* `continue` *Statement*

Click here to view code image

```
class Skip {
  public static void main(String[] args) {
    System.out.println("i    sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4) continue;              // (1) Control to (2).
      // Rest of loop body skipped when i has the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
      // (2) Continue with update expression in the loop header.
    } // end for
  }
}
```

Output from the program:

```
i    sqrt(i)
1    1.00
2    1.41
3    1.73
5    2.24
```

A labeled `continue` statement must occur within a labeled loop that has the same label. Execution of the labeled `continue` statement then transfers control to the end of that enclosing labeled loop. In Example 4.13, the unlabeled `continue` statement at (3) transfers control to (5) when it is executed; that is, the rest of the loop body is skipped and execution continues with the update expression in the inner loop. The labeled `continue` statement at (4) transfers control to (6) when it is executed; that is, it terminates the inner loop but execution continues with the update expression in the loop labeled `outer`. It is instructive to compare the output from Example 4.11 (labeled `break`) with that from Example 4.13 (labeled `continue`).

Example 4.13 *Labeled* `continue` *Statement*

Click here to view code image

```
class LabeledSkip {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
```

```
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) continue;                             // (3) Control to
          System.out.println("Element[" + i + ", " + j + "]: " +
              squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) continue outer;                     // (4) Control to
          // (5) Continue with update expression in the inner loop header.
        } // end inner loop
        // (6) Continue with update expression in the outer loop header.
      } // end outer loop
    System.out.println("sum: " + sum);
  }
}
```

Output from the program:

```
Element[0, 1]: 3
Element[0, 2]: 5
Element[1, 0]: 2
Element[1, 2]: 6
Element[2, 0]: 9
sum: 25
```

4.13 The `return` Statement

The `return` statement is used to stop execution of a method (or a constructor) and transfer control back to the calling code (also called the *caller* or *invoker*). The usage of the two forms of the `return` statement is dictated by whether that statement is used in a `void` or a non-`void` method (Table 4.2). The first form does not return any value to the calling code, but the second form does. Note that the keyword `void` does not represent any type.

In Table 4.2, the *expression* must evaluate to a primitive value or a reference value, and its type must be *assignable* to the *return type* specified in the method header (§2.7, p. 54, and §5.9, p. 261). See also the discussion on covariant return in connection with method overriding in §5.1, p. 201.

As can be seen from Table 4.2, a `void` method need not have a `return` statement—in which case the control typically returns to the caller after the last statement in the method body has been executed. However, a `void` method can specify only the first form of the `return` statement. This form of the `return` statement can also be used in constructors, as they likewise do not return a value.

Table 4.2 also shows that the first form of the `return` statement is not allowed in a non-`void` method. The second form of the return statement is mandatory in a non-`void` method, if the method execution is not terminated programmatically—for example, by throwing an exception. Example 4.14 illustrates the use of the `return` statement summarized in Table 4.2. A recommended best practice is to document the value returned by a method in a Javadoc comment using the `@return` tag.

Table 4.2 *The* `return` *Statement*

Form of `return` statement	**In `void` method or in constructor**	**In non-`void` method**
`return;`	Optional	Not allowed
`return` *expression*;	Not allowed	Mandatory, if the method is not terminated explicitly

Example 4.14 *The* `return` *Statement*

Click here to view code image

```
public class ReturnDemo {

  public static void main (String[] args) { // (1) void method can use return.
    if (args.length == 0) return;
    output(checkValue(args.length));
  }

  static void output(int value) {  // (2) void method need not use return.
    System.out.println(value);
    return 'a';                    // Not OK. Cannot return a value.
  }

  static int checkValue(int i) {   // (3) Non-void method: Any return statemen
                                   //     must return a value.
    if (i > 3)
      return i;                    // OK.
    else
      return 2.0;                  // Not OK. double not assignable to int.
  }

  static int absentMinded() {      // (4) Non-void method.
    throw new RuntimeException();  // OK: No return statement provided, but
                                   // method terminates by throwing an excepti
  }
}
```

Review Questions

4.5 What will be the result of attempting to compile and run the following code?

Click here to view code image

```
class MyClass {
  public static void main(String[] args) {
    boolean b = false;
    int i = 1;
    do {
      i++;
      b = ! b;
    } while (b);
    System.out.println(i);
  }
}
```

Select the one correct answer.

a. The code will fail to compile because `b` is an invalid condition for the `do-while` statement.

b. The code will fail to compile because the assignment `b = ! b` is not allowed.

c. The code will compile without error and will print `1` at runtime.

d. The code will compile without error and will print `2` at runtime.

e. The code will compile without error and will print `3` at runtime.

4.6 What will be the output when running the following program?

Click here to view code image

```
public class StillMyClass {
  public static void main(String[] args) {
    int i = 0;
    int j;
    for (j = 0; j < 10; ++j) { i++; }
    System.out.println(i + " " + j);
  }
}
```

Select the two correct answers.

a. The first number printed will be `9`.

b. The first number printed will be `10`.

c. The first number printed will be `11`.

d. The second number printed will be `9`.

e. The second number printed will be `10`.

f. The second number printed will be `11`.

4.7 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
class AnotherClass {
  public static void main(String[] args) {
    int i = 0;
    for (; i < 10; i++) ;       // (1)
    for (i = 0;; i++) break;    // (2)
    for (i = 0; i < 10;) i++;   // (3)
    for (;;) ;                  // (4)
    }
  }
```

Select the one correct answer.

a. The code will fail to compile because of errors in the `for` loop at (1).
b. The code will fail to compile because of errors in the `for` loop at (2).
c. The code will fail to compile because of errors in the `for` loop at (3).
d. The code will fail to compile because of errors in the `for` loop at (4).
e. The code will compile without error, and the program will run and terminate without any output.
f. The code will compile without error, but will never terminate when run.

4.8 Given the following code fragment, which of the following lines will be a part of the output?

Click here to view code image

```
outer:
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 2; j++) {
    if (i == j) {
      continue outer;
    }
    System.out.println("i=" + i + ", j=" + j);
  }
}
```

Select the two correct answers.

a. `i=1, j=0`
b. `i=0, j=1`
c. `i=1, j=2`
d. `i=2, j=1`
e. `i=2, j=2`
f. `i=3, j=3`
g. `i=3, j=2`

4.9 Which declarations, when inserted at (1), will result in the program compiling and printing `90` at runtime?

Click here to view code image

```
public class RQ400A10 {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    int sum = 0;
    for (int i : nums)
      sum += i;
    System.out.println(sum);
  }
}
```

Select the two correct answers.

a. `Object[] nums = {20, 30, 40};`

b. `Number[] nums = {20, 30, 40};`

c. `Integer[] nums = {20, 30, 40};`

d. `int[] nums = {20, 30, 40};`

e. None of the above

4.10 Which method declarations, when inserted at (1), will result in the program compiling and printing `90` when run?

Click here to view code image

```
public class RQ400A30 {
  public static void main(String[] args) {
    doIt();
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

Select the two correct answers.

a. Click here to view code image

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  for (int sum = 0, i : nums)
    sum += i;
  System.out.println(sum);
}
```

b. Click here to view code image

```
public static void doIt() {
  for (int sum = 0, i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

c. Click here to view code image

```
public static void doIt() {
  int sum = 0;
  for (int i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

d. Click here to view code image

```
public static void doIt() {
  int sum = 0;
  for (int i : new int[] {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

e. Click here to view code image

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  int sum = 0;
  for (int i : nums)
    sum += i;
  System.out.println(sum);
}
```

4.11 Which of the following statements are true about the following `for(:)` loop?

Click here to view code image

```
for (type variable : expression) statement
```

Select the three correct answers.

a. The *variable* is only accessible in the `for(:)` loop body.

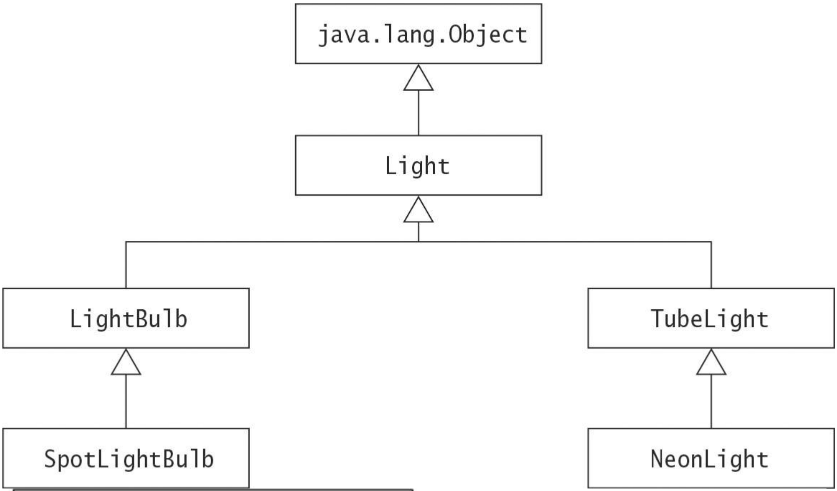
b. The *expression* is only evaluated once.

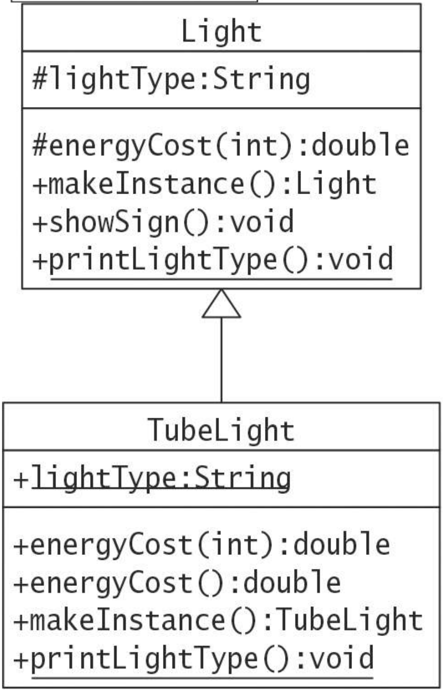
c. The type of the expression must be `java.lang.Iterable<E>` or an array type.

d. Changing the value of the *variable* in the loop body affects the data structure represented by the *expression*.

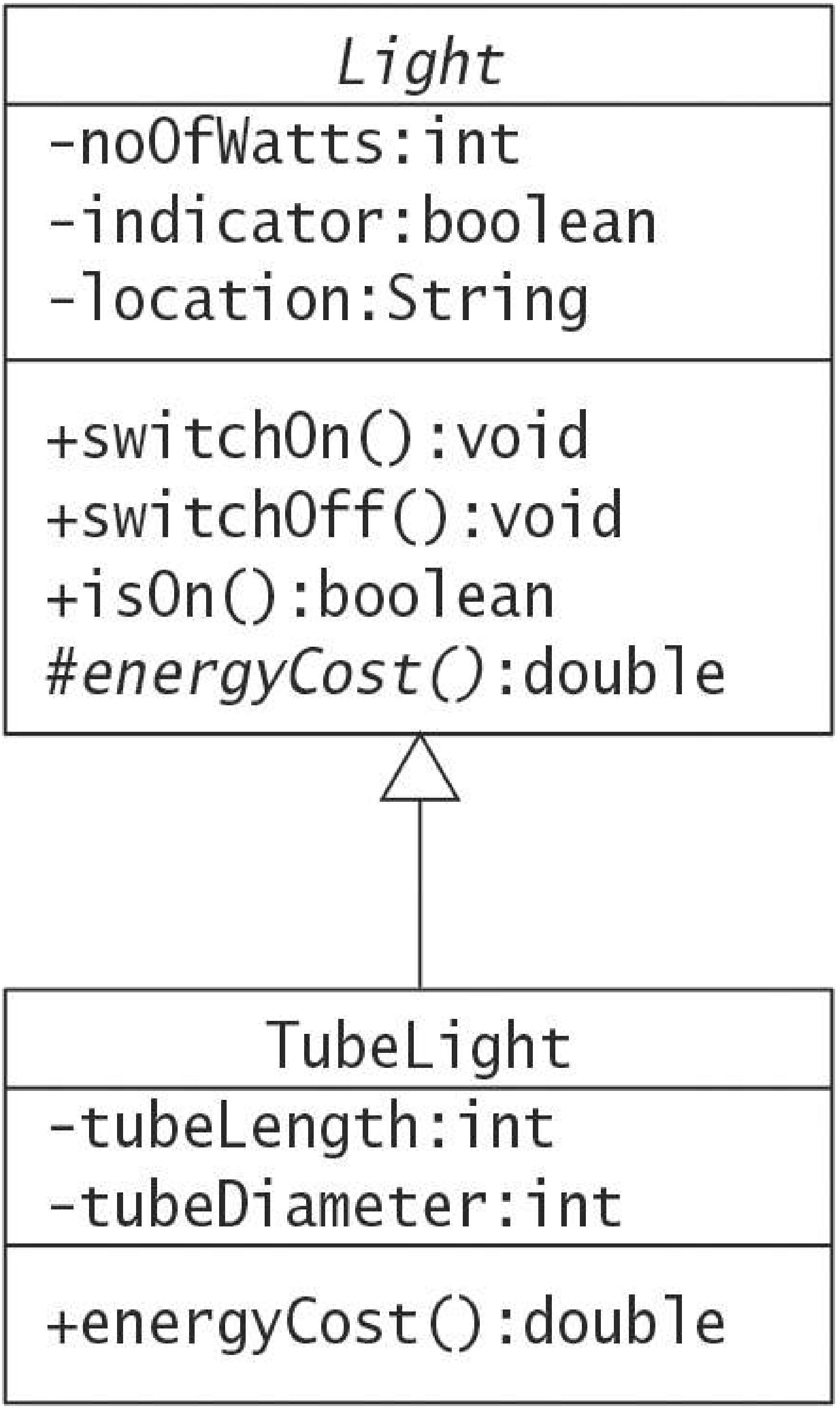
e. The loop runs backward if the *expression* is negated as follows: `!`*expression*.

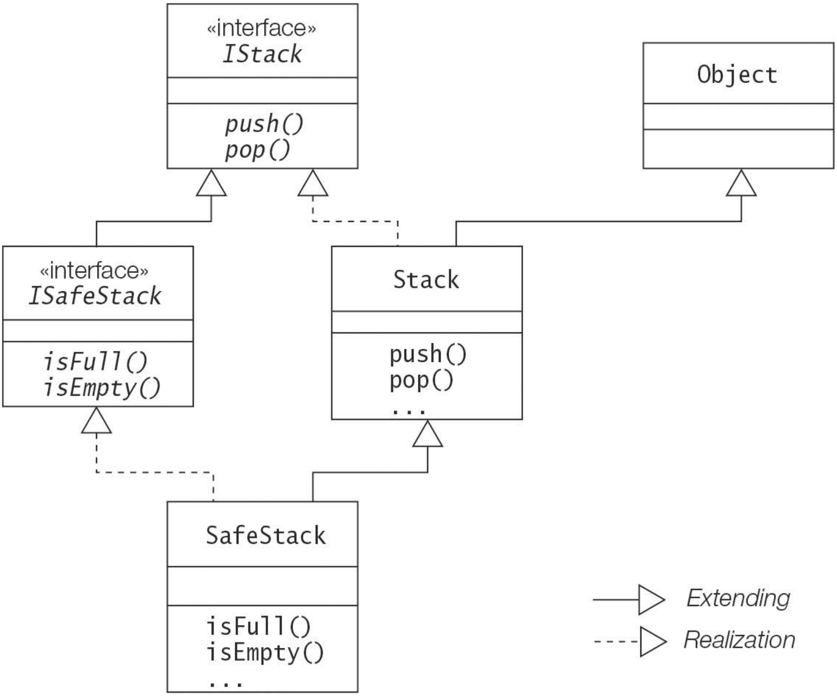
f. We can iterate over several data structures simultaneously in a `for(:)` loop.

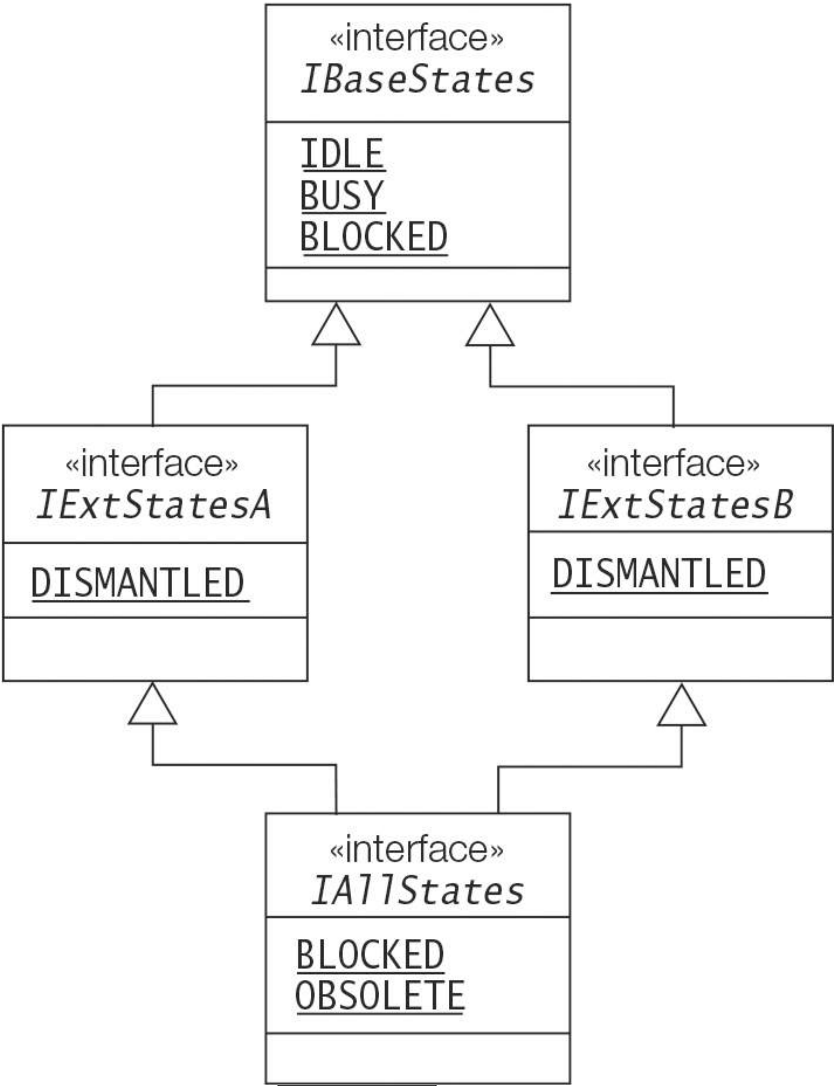

java.lang.Object
Light
LightBulb
TubeLight
SpotLightBulb
NeonLight

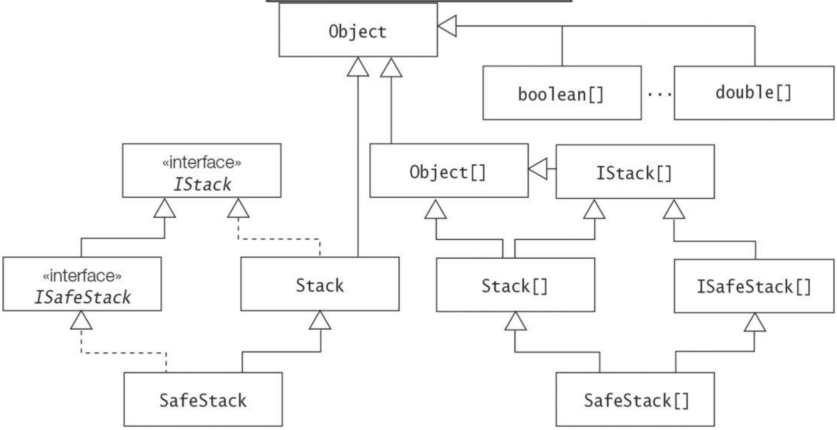

Light
#lightType:String
#energyCost(int):double
+makeInstance():Light
+showSign():void
+printLightType():void
TubeLight
+lightType:String
+energyCost(int):double
+energyCost():double
+makeInstance():TubeLight
+printLightType():void

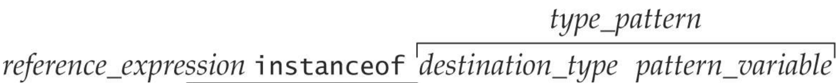
Accessibility:
+ : public
- : private
# : protected
None : package

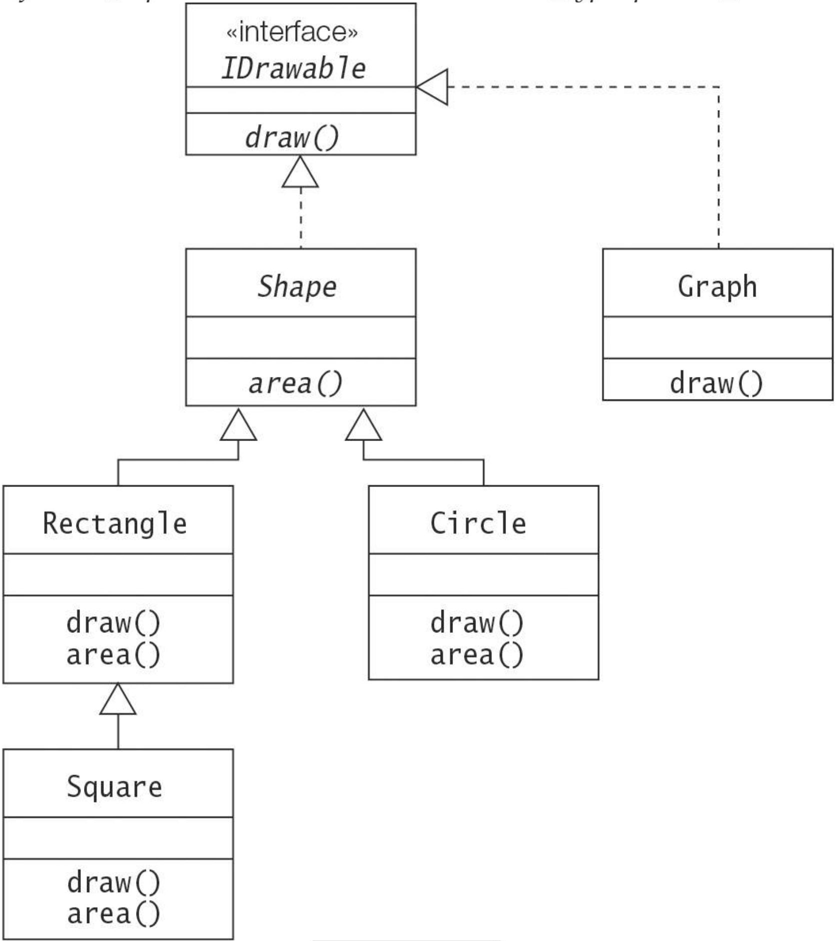

Light
-noOfWatts:int
-indicator:boolean
-location:String
+switchOn():void
+switchOff():void
+isOn():boolean
#*energyCost()*:double
TubeLight
-tubeLength:int
-tubeDiameter:int
+energyCost():double

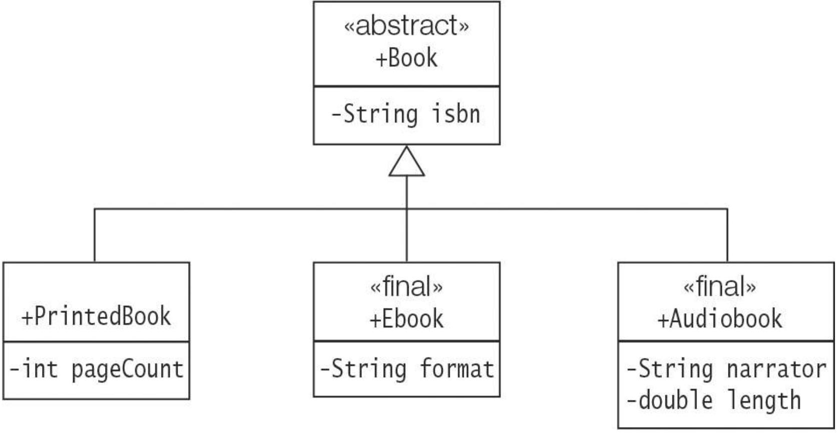

«interface»
IStack
push()
pop()
Object
«interface»
ISafeStack
isFull()
isEmpty()
Stack
push()
pop()
...
SafeStack
isFull()
isEmpty()
...
Extending
Realization

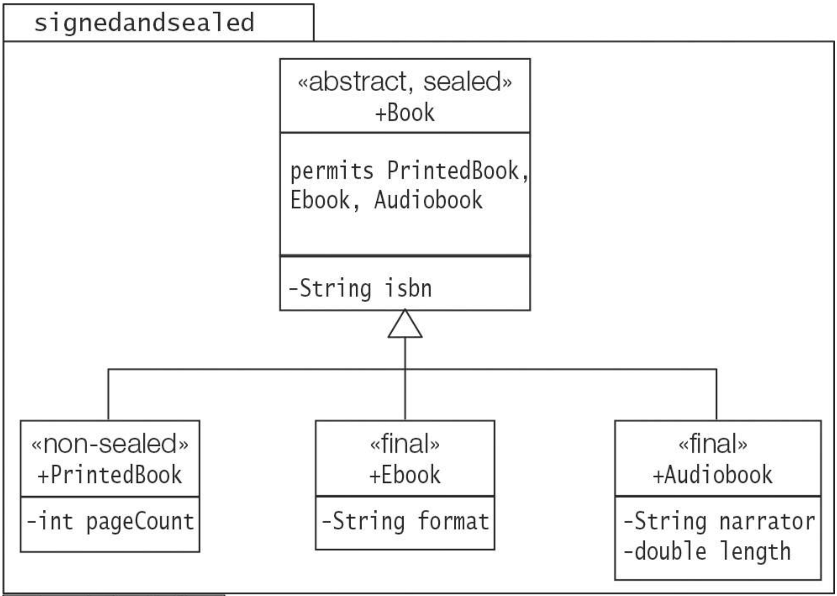

«interface»
IBaseStates
IDLE
BUSY
BLOCKED
«interface»
IExtStatesA
DISMANTLED
«interface»
IExtStatesB
DISMANTLED
«interface»
IAllStates
BLOCKED
OBSOLETE

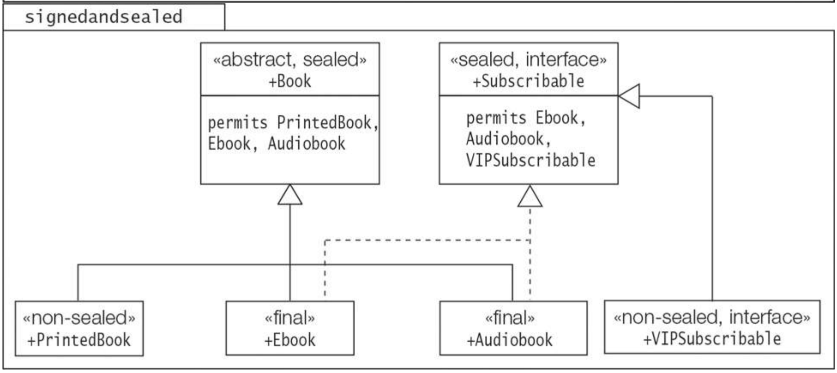

Object
boolean[]
...
double[]
«interface»
IStack
Object[]
IStack[]
«interface»
ISafeStack
Stack
Stack[]
ISafeStack[]
SafeStack
SafeStack[]


type_pattern
reference_expression instanceof destination_type pattern_variable


«interface»
IDrawable
draw()
Shape
area()
Graph
draw()
Rectangle
draw()
area()
Circle
draw()
area()
Square
draw()
area()


«abstract»
+Book
-String isbn
+PrintedBook
-int pageCount
«final»
+Ebook
-String format
«final»
+Audiobook
-String narrator
-double length


signedandsealed
«abstract, sealed»
+Book
permits PrintedBook, Ebook, Audiobook
-String isbn
«non-sealed»
+PrintedBook
-int pageCount
«final»
+Ebook
-String format
«final»
+Audiobook
-String narrator
-double length


signedandsealed
«abstract, sealed»
+Book
permits PrintedBook, Ebook, Audiobook
«sealed, interface»
+Subscribable
permits Ebook, Audiobook, VIPSubscribable
«non-sealed»
+PrintedBook
«final»
+Ebook
«final»
+Audiobook
«non-sealed, interface»
+VIPSubscribable

Object-Oriented Programming

5

Chapter Topics

- Implementing inheritance and its implications in object-oriented programming
- Implications of the subtype–supertype relationship
- Differentiating between inheritance (*is-a*) and aggregation (*has-a*)
- Distinguishing between static and dynamic types of a reference
- Overriding and hiding superclass members
- Understanding method overriding and comparing it with method overloading
- Using the `super` reference to access superclass members
- Using the `this()` and `super()` calls for constructor chaining
- Declaring `abstract` classes and methods
- Using the `final` modifier to declare classes, methods, fields, and local variables
- Declaring interfaces, and the implications of multiple interface inheritance
- Understanding the implications of subtyping for reference arrays
- Understanding conversions when assigning, casting, and passing references
- Understanding resolution of overloaded methods
- Identifying the type of objects and pattern matching using the `instanceof` operator
- Taking advantage of polymorphism and how dynamic method lookup works
- Declaring and using enum types
- Declaring and using record classes
- Declaring and using sealed classes and interfaces

Java SE 17 Developer Exam O	
[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ❍ *Record classes are covered in this chapter.* ❍ *For creating normal classes and defining class members and constructors, see Chapter 3, p. 97.* ❍ *For instance and static initializers, see Chapter 10.5, p. 540.*	*§5.14, p. 299*
[3.3] Implement overloading, including var-arg methods ❍ *Comparing overloading and overriding is covered in this chapter.* ❍ *For overloading methods, including varargs methods, see Chapter 3, p. 97.*	*§5.3, p. 202*
[3.5] Implement inheritance, including abstract and sealed classes. Override methods, including that of Object class. Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching ❍ *Inheritance, abstract and sealed classes, overriding methods, polymorphism and type casting, static and dynamic types of a reference, and* `instanceof` *type comparison and pattern match operators are covered in this chapter.* ❍ *For overriding methods of the* `Object` *class, see §14.1, p. 743.*	*§5.1, p. 191* *to* *§5.4, p. 218* *§5.11, p. 269* *§5.12, p. 278* *§5.15, p. 311*
[3.6] Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods ❍ *Only nonfunctional interfaces are covered in this chapter.* ❍ *For functional interfaces, see §13.1, p. 675.*	*§5.6, p. 237*
[3.7] Create and use enumerations with fields, methods and constructors	*§5.13, p. 287*
Java SE 11 Developer Exam O	
[3.5] Create and use subclasses and superclasses, including abstract classes	*§5.1, p. 191* *to* *§5.4, p. 218*
[3.6] Utilize polymorphism and casting to call methods, differentiating object type versus reference type	*§5.11, p. 269* *§5.12, p. 278*
[3.7] Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods ❍ *Only nonfunctional interfaces are covered in this chapter.*	*§5.6, p. 237*

○ *For functional interfaces, see §13.1, p. 675.*

[3.8] Create and use enumerations	§5.13, p. 287

This chapter delves into two of the main pillars of object-oriented programming (OOP): inheritance and polymorphism. The discussion of the other two OOP concepts, abstraction and encapsulation, can be found in §1.2, p. 5, and §6.1, p. 324, respectively.

Many Java features relating to OOP are covered in this chapter: subtype–supertype relationship, abstract classes and methods, final declarations, interfaces, reference conversions, enum types, record classes, and sealed classes and interfaces. There is a lot of ground to cover in this chapter, so take a deep breath and dive in.

5.1 Implementing Inheritance

Inheritance is one of the fundamental mechanisms for code reuse in OOP. It allows new classes to be derived from existing ones. The new class (also called a *subclass*, *subtype*, *derived class*, or *child class*) can inherit members from the old class (also called a *superclass*, *supertype*, *base class*, or *parent class*). The subclass can add new behavior and properties and, under certain circumstances, modify its inherited behavior.

A subclass specifies the name of its superclass in the subclass header using the `extends` clause.

Click here to view code image

```
class TubeLight extends Light { ... }   // TubeLight is a subclass of Light.
```

The subclass specifies only the additional new and modified members in its class body. The rest of its declaration is made up of its inherited members. If no `extends` clause is specified in the header of a class declaration, the class implicitly inherits from the `java.lang.Object` class. Selected methods inherited by all objects from the `Object` class are covered in connection with object comparison (Chapter 14, p. 741) and thread synchronization (§22.4, p. 1396).

Inheritance of members is closely tied to their declared *accessibility*. If a superclass member is accessible by its simple name in the subclass (without the use of any extra syntax, like `super`), that member is considered inherited. Conversely, private, overridden, and hidden members of the superclass are *not* inherited. Inheritance should not be confused with the *existence* of such members in the state of a subclass object.

The declaration of the `Light` class at (1) in Example 5.1 implicitly extends the `java.lang.Object` class, as no explicit `extends` clause is specified. Also in Example 5.1, the subclass `TubeLight` at (2) explicitly uses the `extends` clause and specifies only members other than those that it already inherits from the superclass `Light` (which, in turn, inherits from the `Object` class). Members of the superclass `Light`, which are accessible by their simple names in the subclass `TubeLight`, are inherited by the subclass, as evident from the output in Example 5.1.

Members of the superclass that have `private` access are not inherited by the subclass and can only be accessed indirectly. The `private` field `indicator` of the superclass `Light` is not inherited, but exists in the subclass object and is indirectly accessible through `public` methods.

Using appropriate access modifiers, the superclass can limit which members can be accessed directly, and therefore, inherited by its subclasses. As shown in Example 5.1, the subclass can use the inherited members as if they were declared in its own class body. This is not the case for members that are declared as `private` in the superclass, as shown at (3), (4), and (5). Members that have package accessibility in the superclass are also not inherited by subclasses in other packages, as these members are accessible by their simple names only in subclasses within the same package as the superclass.

Since constructors are *not* members of a class, they are *not* inherited by a subclass.

Example 5.1 *Extending Classes: Inheritance and Accessibility*

Click here to view code image

```
// File: Utility.java
class Light {                            // (1)
  // Instance fields:
            int     noOfWatts;           // Wattage
  private   boolean indicator;           // On or off
  protected String  location;            // Placement

  // Static field:
  private static int counter;            // Number of Light objects created

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    ++counter;                          // Increment counter.
  }

  // Instance methods:
  public  void    switchOn()  { indicator = true; }
  public  void    switchOff() { indicator = false; }
  public  boolean isOn()      { return indicator; }
  private String  getLocation() { return location; }

  // Static methods:
  public static int getCount() { return counter; }
}
//____________________________________________________________________________
class TubeLight extends Light {         // (2) Subclass uses the extends claus
  // Instance fields:
  private int tubeLength;               // Length in millimeters
  private int tubeDiameter;             // Diameter in millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in supercla
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }
  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("From the subclass:");
    System.out.println("Tube length (mm): "  + getTubeLength());
    System.out.println("Tube diameter (mm): " + tubeDiameter);
    System.out.println();
    System.out.println("From the superclass:");
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // (3) Not inherited
    System.out.println("Location: "     + location);      // Inherited.
//  System.out.println("Counter: "   + counter);          // (4) Not inherited
    switchOn();                                           // Inherited

    switchOff();                                          // Inherited
    System.out.println("Indicator: "    + isOn());        // Inherited.
//  System.out.println("Location: " + getLocation());     // (5) Not inherited
    System.out.println("Number of lights: " + getCount());// Inherited.

  }
}
//____________________________________________________________________________
public class Utility {
  public static void main(String[] args) {
    TubeLight loftLight = new TubeLight(18, true, "Loft", 590, 26);
    loftLight.printInfo();
  }
}
```

Output from the program:

Click here to view code image

```
From the subclass:
Tube length (mm): 590
Tube diameter (mm): 26
```

```
From the superclass:
Wattage: 18
Location: Loft
Indicator: false
Number of lights: 1
```

In Java, a class can extend only *one* class; that is, it can have only one direct superclass. This kind of inheritance is sometimes called *single* or *linear implementation inheritance*. The name is appropriate, as the subclass inherits the *implementation* of its superclass. Java only allows *single inheritance of implementation* using the `extends` clause. *Multiple inheritance of implementation* occurs when a class inherits multiple implementations from the interfaces it implements, but this is *not* allowed in Java (p. 240).

The inheritance relationship can be depicted as an *inheritance hierarchy* (also called a *class hierarchy*), where each subclass is connected by the inheritance arrow to its direct superclass. The `java.lang.Object` class is always at the top (the *root*) of any Java inheritance hierarchy, as all classes extend (either directly or indirectly) this class. The path from a subclass in the inheritance hierarchy to the root traces all the superclasses that the subclass inherits from. Classes up in the hierarchy are more *generalized* (often called *broader*), as they abstract the class behavior. Classes lower in the hierarchy are more *specialized* (often called *narrower*), as they customize the inherited behavior by additional properties and behavior. Figure 5.1 illustrates the inheritance relationship between the class `Light`, which represents the more general abstraction, and its more specialized subclasses. The class `SpotLightBulb` inherits from the classes `LightBulb`, `Light`, and `Object`.

Figure 5.1 *Inheritance Hierarchy*

Relationships: *is-a* and *has-a*

The inheritance relationship between a subclass and its superclass is embodied by the *is-a* relationship. Since a subclass inherits from its superclass, a subclass object *is-a* superclass object, and can be used wherever an object of the superclass can be used. It has particular consequences for how objects can be used. An object of the `TubeLight` class *is-an* object of the superclass `Light`. Referring to Figure 5.1, an object of the `TubeLight` class can be used wherever an object of the superclass `Light` can be used.

The inheritance relationship is *transitive*: If class `B` extends class `A` and class `C` extends class `B`, then class `C` will also inherit from class `A` via class `B`. In Figure 5.1, an object of the `SpotLightBulb` class *is-an* object of the class `Light`. The *is-a* relationship does not hold between peer classes: An object of the `LightBulb` class is *not* an object of the class `TubeLight`, and vice versa.

Whereas inheritance defines the relationship *is-a* between a superclass and its subclasses, *aggregation* defines the relationship *has-a* (also called the *whole–part* relationship) between an instance of a class and its constituents (also called *parts)*. Aggregation comprises the *usage* of objects.

A class declaration with instance field references implements an aggregate object. The class `Light` is declared with three instance fields: an instance field to store its wattage (`noOfWatts`), an instance field to store whether it is on or off (`indicator`), and an instance field reference (`location`) to a `String` object to store its location (in fact, its only constituent object). An instance of class `Light` *has* (or *uses*) an object of class `String`. In Java, a composite object cannot contain other objects. It can only store *reference values* of its constituent objects in its fields. This relationship defines an *aggregation hierarchy* (also called *object hierarchy*) that embodies the *has-a* relationship. Constituent objects can be shared between objects. If their lifetimes are dependent on the lifetime of the aggregate object, then this relationship is called *composition*, and implies strong ownership of the parts by the composite object.

Choosing between inheritance and aggregation to model relationships can be a crucial design decision. A good design strategy advocates that inheritance should be used only if the relationship *is-a* is unequivocally maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice. A *role* is often confused with an *is-a* relationship. For example, given the class `Employee`, it would not be a good idea to model the roles that an employee can play (such as manager or cashier) by inheritance, if these roles change intermittently. Changing roles would involve a new object to represent the new role every time this happens.

Code reuse is also best achieved by aggregation when there is no *is-a* relationship. Enforcing an artificial *is-a* relationship that is not naturally present is usually not a good idea. Methods that contradict the abstraction represented by the subclass can be invoked. Using aggregation in such a case results in a better solution.

Both inheritance and aggregation promote encapsulation of *implementation*. Changes in the implementation of constituent objects generally have minimal impact on the clients of the composite object, since these clients do not directly deal with the underlying objects. However, changing the *contract* of a superclass can have consequences for the subclasses (called the *ripple effect*) as well as for clients that are dependent on a particular behavior of the subclasses. For this reason, aggregation provides stronger encapsulation than inheritance.

The Subtype–Supertype Relationship

A class defines a *reference type*, a data type whose objects can be accessed only by references. Therefore, the inheritance hierarchy can be regarded as a *type hierarchy*, embodying the *subtype–supertype relationship* between reference types. The *subclass–superclass* relationship is a special case of the subtype–supertype relationship that is between classes. The subclass–superclass relationship allows *single inheritance of type*, meaning that the subclass inherits the *type* of its direct superclass. This is in contrast to a class that implements several interfaces, resulting in *multiple inheritance of type*—that is, a class inherits the *type* of all interfaces it implements.

In the context of Java, the subtype–supertype relationship implies that the reference value of a subtype object can be assigned to a supertype reference because a subtype object can be substituted for a supertype object. This assignment involves a *widening reference conversion*, as references are assigned *up* the inheritance hierarchy. Using the reference types in Example 5.1, the following code assigns the reference value of an object of the subtype `TubeLight` to the reference `light` of the supertype `Light`:

Click here to view code image

```
Light light = new TubeLight(36, false, "Basement",
                            1200, 26);      // (1) widening reference conversi
```

An implicit widening conversion takes place under assignment, as the reference value of a narrower type (subtype `TubeLight`) object is being assigned to a reference of a broader type (supertype `Light`). We can now use the reference `light` to invoke those methods on the subtype object that are inherited from the supertype `Light`:

Click here to view code image

```
light.switchOn();                              // (2)
```

Note that the compiler only knows about the *static type* (also called the *declared type*) of the reference `light`, which is `Light`, and ensures that only methods from this type can be called using the reference `light`. However, at runtime, the reference `light` will refer to an object of the subtype `TubeLight` when the call to the method `switchOn()` is executed. It is the *type of the object* that the reference refers to at runtime that determines which method is executed. The subtype object inherits the `switchOn()` method from its supertype `Light`, so this method is executed. The type of the object that the reference refers to at runtime is called the *dynamic type* (a.k.a. the *runtime type* or the *object type*) of the reference.

One might be tempted to invoke methods exclusive to the `TubeLight` subtype via the supertype reference `light`:

Click here to view code image

```
light.getTubeLength();                         // (3) Compile-time error!
```

This code will not work, as the compiler does not know which object the reference `light` will denote at runtime; it only knows the declared type of the reference. As the declaration of the class `Light` does not have a method called `getTubeLength()`, this method call at (3) results in a compile-time error. Eliciting subtype-specific behavior using a supertype reference requires a *narrowing reference conversion* with an explicit cast (p. 278).

Overriding Instance Methods

Under certain circumstances, a subclass can *override instance methods* from its superclass. Overriding such a method allows the subclass to provide its *own* implementation of the method. The overridden method in the superclass is *not* inherited by the subclass. When the method is invoked on an object of the subclass, it is the method implementation in the subclass that is executed. The new method in the subclass must abide by the following rules of method overriding:

- The new method definition in the subclass must have the same *method signature*. In other words, the method name, and the types and the number of parameters, including their order, must be the same as in the overridden method of the superclass.

Whether parameters in the overriding method should be `final` is at the discretion of the subclass. A method's signature does not comprise the `final` modifier of parameters, only their types and order.

- The return type of the overriding method can be a *subtype* of the return type of the overridden method (called *covariant return,* p. 201).
- The new method definition cannot *narrow* the accessibility of the method, but it can *widen* it.
- The new method definition can throw either all or none, or a subset of the *checked* exceptions (including their subclasses) that are specified in the `throws` clause of the overridden method in the superclass (§7.5, p. 388)—that is, the overriding method may not throw any checked exception that is *wider* than the checked exceptions thrown by the overridden method. However, the overriding method may throw any *unchecked* exceptions.

The criteria for overriding methods also apply to interfaces, where a subinterface can override `abstract` and `default` method declarations from its superinterfaces (p. 237).

Figure 5.2 *Inheritance Hierarchy for Example 5.2*

The canonical examples of method overriding in Java are the `equals()`, `hashCode()`, and `toString()` methods of the `Object` class. The wrapper classes and the `String` class all override these methods (Chapter 8, p. 423). Overriding these methods in user-defined classes is thoroughly covered in connection with object comparison (Chapter 14, p. 741).

Example 5.2 illustrates overriding, overloading, and hiding of members in a class. Figure 5.2 gives an overview of the two main classes in Example 5.2. The new definition of the `energyCost()` method at (7) in the subclass `TubeLight` has the same signature and the same return type as the method at (2) in the superclass `Light`. The new definition specifies a subset of the exceptions (`ZeroHoursException`) thrown by the overridden method (the exception class `InvalidHoursException` is a superclass of `NegativeHoursException` and `ZeroHoursException`). The new definition also widens the accessibility (`public`) from what it was in the overridden definition (`protected`). The overriding method declares the parameter to be `final`, but this has no bearing on overriding the method.

Click here to view code image

```
// The overridden method in the superclass Light:
protected double energyCost(int noOfHours)      // (2) Instance method
    throws InvalidHoursException { ... }

// The overriding method in the subclass TubeLight:
@Override
public double energyCost(final int noOfHours)   // (7) Overriding instance
    throws ZeroHoursException { ... }           //     method at (2).
```

The astute reader will have noticed the `@Override` annotation preceding the method definition at (7). The compiler will now report an error if the method definition at (7) does *not* override an inherited method. The annotation helps to ensure that the method definition overrides the inherited method, rather than overloading another method silently.

Invocation of the method `energyCost()` on an object of subclass `TubeLight` using references of the subclass and the superclass at (15) and (16), respectively, results in the new definition at (7) being executed, since both references are aliases of the `TubeLight` object created at (12).

Click here to view code image

```
tubeLight.energyCost(50);                     // (15) Invokes method at (7).
light1.energyCost(50);                        // (16) Invokes method at (7).
```

Not surprisingly, the invocation of the method `energyCost()` on an object of superclass `Light`, using a reference of the superclass at (17), results in the overridden definition at (2) being executed:

Click here to view code image

```
light2.energyCost(50);                        // (17) Invokes method at (2).
```

Example 5.2 *Overriding, Overloading, and Hiding*

Click here to view code image

```
// File: Client2.java
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
```

```
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                        // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {           // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//____________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//____________________________________________________________________________

public class Client2 {
  public static void main(String[] args)          // (11)
      throws InvalidHoursException {
```

```
    TubeLight tubeLight = new TubeLight();        // (12)
    Light     light1    = tubeLight;              // (13) Aliases.
    Light     light2    = new Light();            // (14)

    System.out.println("Invoke overridden instance method:");
    tubeLight.energyCost(50);                     // (15) Invokes method at (7
    light1.energyCost(50);                        // (16) Invokes method at (7
    light2.energyCost(50);                        // (17) Invokes method at (2

    System.out.println(
        "\nInvoke overridden instance method with covariant return:");
    System.out.println(
        light2.makeInstance().getClass());        // (18) Invokes method at (3
    System.out.println(
        tubeLight.makeInstance().getClass());     // (19) Invokes method at (9

    System.out.println("\nAccess hidden field:");
    System.out.println(tubeLight.lightType);      // (20) Accesses field at (6
    System.out.println(light1.lightType);         // (21) Accesses field at (1
    System.out.println(light2.lightType);         // (22) Accesses field at (1

    System.out.println("\nInvoke hidden static method:");
    tubeLight.printLightType();                   // (23) Invokes method at (1
    light1.printLightType();                      // (24) Invokes method at (5
    light2.printLightType();                      // (25) Invokes method at (5

    System.out.println("\nInvoke overloaded method:");
    tubeLight.energyCost();                       // (26) Invokes method at (8
  }
}
```

Output from the program:

Click here to view code image

```
Invoke overridden instance method:
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> Light.energyCost(int): Energy cost for Generic Light: 10.0

Invoke overridden instance method with covariant return:
>> Light.makeInstance(): class Light
>> TubeLight.makeInstance(): class TubeLight

Access hidden field:
Tube Light
Generic Light
Generic Light

Invoke hidden static method:
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
>> Static Light.printLightType(): Generic Light

Invoke overloaded method:
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
```

Here are a few more facts to note about overriding.

- A subclass must use the keyword `super` to invoke an overridden method in the superclass.
- A `final` method cannot be overridden because the

modifier `final` prevents method overriding. An attempt to override a `final` method will result in a compile-time error. An `abstract` method, in contrast, requires the non-`abstract` subclasses to override the method so as to provide an implementation. Abstract and final methods are discussed in §5.4, p. 224, and §5.5, p. 226, respectively.

- The access modifier `private` for a method means that the method is not accessible outside the class in which it is defined; therefore, a subclass cannot override it. However, a subclass can give its own definition of such a method, which may have the same signature as the method in its superclass.

- A subclass within the same package as the superclass can override any non-`final` methods declared in the superclass. However, a subclass in a different package can override only the non-`final` methods that are declared as either `public` or `protected` in the superclass.

- An instance method in a subclass cannot override a `static` method in the superclass. The compiler will flag such an attempt as an error. A `static` method is class specific and not part of any object, while overriding methods are invoked on behalf of objects of the subclass. However, a `static` method in a subclass can *hide* a `static` method in the superclass, as we shall see (p. 203), but it cannot hide an instance method in the superclass.

- Constructors, since they are not inherited, cannot be overridden.

Covariant `return` in Overriding Methods

In Example 5.2, the definition of the method `makeInstance()` at (9) overrides the method definition at (3).

Click here to view code image

```
// The overridden method in the superclass Light:
public Light makeInstance() { ... }     // (3) Instance method

// The overriding method in the subclass TubeLight:
@Override
public TubeLight makeInstance() { ... } // (9) Overriding instance method at (
```

Note that the method signatures are the same, but the return type at (9) is a subtype of the return type at (3). The method at (9) returns an object of the subtype `Tube-Light`, whereas the method at (3) returns an object of the supertype `Light`. This is an example of *covariant return*.

Depending on whether we call the method `makeInstance()` on an object of the subtype `TubeLight` or an object of the supertype `Light`, the respective method definition will be executed. The code at (18) and (19) illustrates which object is returned by the method, depending on which method definition is executed.

Note that covariant return applies only to *reference* types, not to primitive types. For example, changing the return type of the `energyCost()` method at (7) to `float` will result in a compile-time error. There is no subtype–supertype relationship between primitive types.

Overriding versus Overloading

Method overriding should not be confused with *method overloading* (§3.6, p. 108). Method overriding always requires the same method signature (name and parameter types) and the same or covariant return types. Overloading occurs when the method names are the same, but the parameter lists differ. Therefore, to overload methods, the parameters must differ in either type, order, or number. As the return type is not a part of the method signature, use of different return types is not sufficient to overload methods. Although methods can be overloaded, as shown at (1) and (2) in Example 5.3, a call to such an overloaded method can be *ambiguous*, as shown at (3). It is not possible to determine which method should be called, (1) or (2). Disambiguation can be done by making the types of the arguments in the call more specific, as shown at (4) and (5).

Example 5.3 *Ambiguous Call to Overloaded Methods*

Click here to view code image

```
public class Overloader {

  static final void callMe(long x, Integer y) {       // (1) Overloaded by (2)
    System.out.println("long, Integer");
  }
  static void callMe(Integer x, long y) {       // (2) Overloaded by (1)
    System.out.println("Integer, long");
  }

  public static void main(String[] args) {
//  callMe(20, 17);                // (3) Ambiguous call: Box 1st or 2nd argum
    callMe(20, Integer.valueOf(17));   // (4) Calls (1): long, Integer
    callMe(Integer.valueOf(20), 17);   // (5) Calls (2): Integer, long
  }
}
```

Output from the program:

```
long, Integer
Integer, long
```

Only non-`final` instance methods in the superclass that are directly accessible from the subclass using their simple name can be overridden. In contrast, both non-`private` instance and `static` methods can be overloaded in the class they are defined in or are in a subclass of the class they are defined in.

Depending on the arguments passed in a call to an overloaded method, the appropriate method implementation is invoked. In Example 5.2, the method `energy-Cost()` at (2) in superclass `Light` is overridden in subclass `TubeLight` at (7) and overloaded at (8). When invoked at (26), the overloaded declaration at (8) is executed.

For overloaded and `static` methods, which method implementation will be executed at runtime is determined at *compile time*. In contrast, for overridden methods, the method implementation to be executed is determined at *runtime* (p. 278).

Hiding Fields

A subclass cannot override inherited *fields* of the superclass, but it can *hide* them. The subclass can define fields with the same name as in the superclass. If this is the case, the fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass. A hidden `static` field can always be accessed by using the superclass name in the subclass declaration. Additionally, the keyword `super` can be used in non-`static` code in the subclass declaration to access hidden `static` fields.

The following distinction between invoking instance methods on an object and accessing fields of an object must be noted. When an instance method is invoked on an object using a reference, it is the *dynamic type* of the reference (i.e., *the type of the current object* denoted by the reference at runtime), not the declared type of the reference, that determines which method implementation will be executed. In Example 5.2 at (15), (16), and (17), this is evident from invoking the overridden method `energyCost()`: The method from the class corresponding to the current object is executed, regardless of the declared reference type.

When a field of an object is accessed using a reference, it is the *declared type* of the reference, not the type of the current object denoted by the reference, that determines which field will actually be accessed. In Example 5.2 at (20), (21), and (22), this is evident from accessing the hidden field `lightType`: The field accessed is the one declared in the class corresponding to the declared reference type, regardless of the object denoted by the reference at runtime.

In contrast to method overriding, where an instance method cannot override a `static` method, there are no such restrictions on the hiding of fields. The field `lightType` is `static` in the subclass, but not in the superclass. The declared type of the fields need not be the same either—only the field name matters in the hiding of fields.

Hiding Static Methods

Only instance methods in an object can be overridden. However, a `static` method in a subclass can *hide* a `static` method from the superclass. Hiding a `static` method is analogous to overriding an instance method except for one important aspect: Calls to `static` methods are *bound at compile time* as opposed to runtime for calls to overridden instance methods, and therefore do *not* exhibit polymorphic behavior exhibited by calls to overridden instance methods (p. 278).

When hiding a `static` method, the compiler will flag an error if the signatures are the same, but the other requirements regarding the return type, `throws` clause, and accessibility are not met. If the signatures are different, the method name is overloaded, not hidden.

A `static` method in the subclass can only hide a `static` method in the superclass. Analogous to an overridden instance method, a hidden superclass `static` method is *not* inherited. Analogous to accessing hidden `static` fields, a hidden `static` method in the superclass can always be invoked by using the superclass name or by using the keyword `super` in non-`static` code in the subclass declaration.

Example 5.2 illustrates invocation of `static` methods using references. Analogous to accessing fields, the `static` method invoked at (23), (24), and (25) is determined by the *declared type* of the reference. At (23), the declared reference type is `TubeLight`;

therefore, the `static` method `printLightType()` at (10) in this class is invoked. At (24) and (25), the declared reference type is `Light`, and the hidden `static` method `print-LightType()` at (5) in that class is invoked. This is borne out by the output from the program.

In summary, a subclass can do any of the following:

- Access inherited members using their simple names
- Override a non-`final` instance method with the same signature as in the superclass
- Overload a `static` or instance method with the same name as in the superclass
- Hide a `static` non-`final` method with the same signature as in the superclass
- Hide a `static` or instance field with the same name as in the superclass
- Declare new members that are not in the superclass
- Declare constructors that can invoke constructors in the superclass implicitly, or use the `super()` construct with the appropriate arguments to invoke them explicitly (p. 209)

Table 5.1 provides a comparison between overriding, hiding, and overloading of methods.

Table 5.1 *Overriding, Hiding, and Overloading of Methods*

Comparison criteria	Overriding instance methods/Hiding static methods	Overloading methods
Method name	Must be the same.	Must be the same.
Argument list	Must be the same.	Must be different.
Return type	Can be the same type or a covariant type.	Can be different.
`throws` clause	Can be restrictive about checked exceptions thrown. Must not throw new checked exceptions, but can include subclasses of exceptions thrown.	Can be different.
Accessibility	Can be less restrictive, but not more restrictive.	Can be different.
`final` modifier	`final` instance/`static` method cannot be overridden/hidden. Compile-time error if `final` instance/`static` method is defined in a subclass.	Can be overloaded.
`private` modifier	`private` instance/`static` method cannot be overridden/hidden. No compile-time error if `private` instance/`static` method is defined in a subclass.	Can be overloaded.
Declaration context	An instance/`static` method can only be overridden/hidden in a subclass.	Can be overloaded in the same class or in a subclass.

Calls to overridden instance methods are bound at runtime, and therefore exhibit polymorphic behavior. [illegible]

Calls to hidden `static` methods are bound at compile time and do *not* exhibit polymorphic behavior.

5.2 The Object Reference `super`

The `this` reference can be used in non-`static` code to refer to the current object (§3.5, p. 106). The keyword `super`, in contrast, can be used in non-`static` code to access fields and invoke methods from the superclass. The keyword `super` provides a reference to the current object as an instance of its superclass. In method invocations with `super`, the method from the superclass is invoked regardless of what the actual type of the current object is or whether the current class overrides the method. This approach is typically used to invoke methods that are overridden and to access members that are hidden to the subclass. Unlike the `this` keyword, the `super` keyword cannot be used as an ordinary reference. For example, it cannot be assigned to other references or cast to other reference types.

Example 5.4 uses the superclass `Light` and its subclass `TubeLight` from Example 5.2, which are also shown in Figure 5.2. In Example 5.4, the class `NeonLight` extends the class `TubeLight`. The declaration of the method `demonstrate()` at (11) in the class `NeonLight` makes use of the `super` keyword to access members higher in its inheritance hierarchy. This is the case when the `showSign()` method is invoked at (12). This method is defined at (4) in the class `Light`, rather than in the direct superclass `TubeLight` of the subclass `NeonLight`. The overridden method `energyCost()` at (7) and its overloaded version at (8) in the class `TubeLight` are invoked, using the object reference `super` at (13) and (14), respectively.

The superclass `Light` has a field named `lightType` and a method named `energyCost` defined at (1) and (2), respectively. One might be tempted to use the syntax `super.super.energyCost(20)` in the subclass `NeonLight` to invoke this method, but this is not a valid construct. One might also be tempted to cast the `this` reference to the class `Light` and try again, as shown at (15). The output shows that the method `energyCost()` at (7) in the class `TubeLight` was executed, not the one from the class `Light`. The reason is that a cast simply changes the type of the reference (in this case to `Light`), not the class of the object (which is still `NeonLight`). Method invocation is determined by the class of the current object, resulting in the inherited method `energyCost()` in the class `TubeLight` being executed. There is no way to invoke the method `energyCost()` in the class `Light` from the subclass `NeonLight`, without declaring a reference of the type `Light`.

At (16), the keyword `super` is used to access the field `lightType` at (6) in the class `TubeLight`, but the keyword `super` is redundant in this case. At (17), the field `light-Type` from the class `Light` is accessed successfully by casting the `this` reference, because it is the type of the reference that determines which field is accessed. From non-`static` code in a subclass, it is possible to directly access fields in a class higher in the inheritance hierarchy by casting the `this` reference. However, it is futile to cast the `this` reference to invoke instance methods in a class higher in the inheritance hierarchy, as illustrated earlier for the overridden method `energyCost()`.

Finally, the calls to the `static` methods at (18) and (19) using the `super` and `this` references, respectively, exhibit runtime behavior analogous to accessing fields, as discussed previously.

Example 5.4 *Using the* `super` *Keyword*

Click here to view code image

```
// File: Client3.java
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                        // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }
  public static void printLightType() {          // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//____________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");

    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//____________________________________________________________________________
class NeonLight extends TubeLight {
  // ...
  public void demonstrate()                       // (11)
      throws InvalidHoursException {
    super.showSign();                             // (12) Invokes method at (4
    super.energyCost(50);                         // (13) Invokes method at (7
    super.energyCost();                           // (14) Invokes method at (8

    ((Light) this).energyCost(50);                // (15) Invokes method at (7

    System.out.println(super.lightType);          // (16) Accesses field at (6
    System.out.println(((Light) this).lightType); // (17) Accesses field at (1

    super.printLightType();                       // (18) Invokes method at (1
    ((Light) this).printLightType();              // (19) Invokes method at (5
  }
}
//____________________________________________________________________________
public class Client3 {
  public static void main(String[] args)
      throws InvalidHoursException {
```

```
      NeonLight neonRef = new NeonLight();
      neonRef.demonstrate();
    }
  }
```

Output from the program:

Click here to view code image

```
>> Light.showSign(): Let there be light!
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
Tube Light
Generic Light
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
```

5.3 Chaining Constructors Using `this()` and `super()`

A basic understanding of constructors (§3.7, p. 109) is beneficent for the discussion in this section.

The `this()` Constructor Call

Constructors cannot be inherited or overridden. They can be overloaded, but only in the same class. Since a constructor always has the same name as the class, each parameter list must be different when defining more than one constructor for a class. In Example 5.5, the class `Light` has three overloaded constructors. In the constructor at (3), the `this` reference is used to access the fields shadowed by the parameters. In the `main()` method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call, as illustrated by the program output.

Example 5.5 *Constructor Overloading*

Click here to view code image

Click here to view code image

```
// File: DemoConstructorCall.java
class Light {
  // Fields:
  private int     noOfWatts;       // wattage
  private boolean indicator;       // on or off
  private String  location;        // placement

  // Constructors:
```

```
  Light() {                                  // (1) No-argument constructor
    noOfWatts = 0;
    indicator = false;
    location  = "X";
    System.out.println("Returning from no-argument constructor no. 1.");
  }
  Light(int watts, boolean onOffState) {                      // (2)
    noOfWatts = watts;
    indicator = onOffState;
    location  = "X";
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//____________________________________________________________________________
public class DemoConstructorCall {
  public static void main(String[] args) {                    // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);
    System.out.println("Creating Light object no. 3.");
    Light light3 = new Light(250, true, "attic");
  }
}
```

Output from the program:

Click here to view code image

```
Creating Light object no. 1.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 2.
Creating Light object no. 3.
Returning from constructor no. 3.
```

Example 5.6 illustrates the use of the `this()` construct, which is used to implement *local chaining* of constructors in the class when an instance of the class is created. The first two constructors at (1) and (2) from Example 5.5 have been rewritten using the `this()` construct in Example 5.6 at (1) and (2), respectively. The `this()` construct can be regarded as being locally overloaded, since its parameters (and hence its signature) can vary, as shown in the body of the constructors at (1) and (2). The `this()` call invokes the local constructor with the corresponding parameter list. In the `main()` method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call when each of the three `Light` objects are created. Calling the no-argument constructor at (1) to create a `Light` object results in the constructors at (2) and (3) being executed as well. This is confirmed by the output from the program.

In this case, the output shows that the constructor at (3) completed first, followed by the constructor at (2), and finally by the no-argument constructor at (1) that was called first. Bearing in mind the definition of the constructors, the constructors are invoked in the *reverse* order; that is, invocation of the no-argument constructor immediately leads to invocation of the constructor at (2) by the call `this(0, false)`, and its invocation leads to the constructor at (3) being called immediately by the call `this(watt, ind, "X")`, with the completion of the execution in the reverse order of their invocation. Similarly, calling the constructor at (2) to create an instance of the `Light` class results in the constructor at (3) being executed as well.

If the `this()` call is specified, it must occur as the *first* statement in a constructor. The `this()` call can be followed by any other relevant code. This restriction is due to Java's handling of constructor invocation in the superclass when an object of the subclass is created. This mechanism is explained in the next subsection.

Example 5.6 *The* `this()` *Constructor Call*

Click here to view code image

```
// File: DemoThisCall.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;

  // Constructors:
  Light() {                                  // (1) No-argument constructor
    this(0, false);
    System.out.println("Returning from no-argument constructor no. 1.");
  }
  Light(int watt, boolean ind) {                              // (2)
    this(watt, ind, "X");
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) { // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//______________________________________________________________________
public class DemoThisCall {
  public static void main(String[] args) {                   // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();                               // (5)
    System.out.println("Creating Light object no. 2.");
```

```
        Light light2 = new Light(250, true);                      // (6)
        System.out.println("Creating Light object no. 3.");
        Light light3 = new Light(250, true, "attic");             // (7)
      }
    }
```

Output from the program:

Click here to view code image

```
Creating Light object no. 1.
Returning from constructor no. 3.
Returning from constructor no. 2.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 3.
Returning from constructor no. 2.
Creating Light object no. 3.
Returning from constructor no. 3.
```

The `super()` Constructor Call

The constructor call `super()` is used in a subclass constructor to invoke a constructor in the *direct* superclass. This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created. A `super()` call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the signature of the call. Since the superclass name is known in the subclass declaration, the compiler can determine the superclass constructor invoked from the signature of the parameter list.

A constructor in a subclass can access the class's inherited members by their simple names. The keyword `super` can also be used in a subclass constructor to access inherited members via its superclass. One might be tempted to use the `super` keyword in a constructor to specify initial values for inherited fields. However, the `super()` construct provides a better solution to initialize the inherited state.

In Example 5.7, the constructor at (3) of the class `Light` has a `super()` call (with no arguments) at (4). Although the constructor is not strictly necessary, as the compiler will insert one—as explained later—it is included here for expositional purposes. The constructor at (6) of the class `TubeLight` has a `super()` call (with three arguments) at (7). This `super()` call will match the constructor at (3) of the superclass `Light`. This is evident from the program output.

Example 5.7 *The* `super()` *Constructor Call*

Click here to view code image

```
// File: ChainingConstructors.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;

  // Constructors:
  Light() {                                  // (1) No-argument constructor
    this(0, false);
    System.out.println(
    "Returning from no-argument constructor no. 1 in class Light");
  }
  Light(int watt, boolean ind) {                              // (2)
    this(watt, ind, "X");
    System.out.println(
    "Returning from constructor no. 2 in class Light");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    super();                                                  // (4)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println(
        "Returning from constructor no. 3 in class Light");
  }
}
//___________________________________________________________________________
class TubeLight extends Light {
  // Instance variables:
  private int tubeLength;
  private int colorNo;

  // Constructors:
  TubeLight(int tubeLength, int colorNo) {                    // (5)
    this(tubeLength, colorNo, 100, true, "Unknown");
    System.out.println(
           "Returning from constructor no. 1 in class TubeLight");
  }
  TubeLight(int tubeLength, int colorNo, int noOfWatts,
            boolean indicator, String location) {             // (6)
    super(noOfWatts, indicator, location);                    // (7)
    this.tubeLength = tubeLength;
    this.colorNo    = colorNo;
    System.out.println(
           "Returning from constructor no. 2 in class TubeLight");
  }
}
//___________________________________________________________________________
public class ChainingConstructors {
  public static void main(String[] args) {
    System.out.println("Creating a TubeLight object.");
    TubeLight tubeLightRef = new TubeLight(20, 5);            // (8)
  }
}
```

Output from the program:

Click here to view code image

```
Creating a TubeLight object.
Returning from constructor no. 3 in class Light
Returning from constructor no. 2 in class TubeLight
Returning from constructor no. 1 in class TubeLight
```

The `super()` construct has the same restrictions as the `this()` construct: If used, the `super()` call must occur as the *first* statement in a constructor, and it can only be used in a constructor declaration. This implies that `this()` and `super()` calls cannot both occur in the same constructor. The `this()` construct is used to *chain* constructors in the *same* class. The constructor at the end of such a chain can invoke a superclass constructor using the `super()` construct. Just as the `this()` construct leads to chaining of constructors in the same class, so the `super()` construct leads to chaining of subclass constructors to superclass constructors. This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way to the top of the inheritance hierarchy, which is always the `Object` class. Note that the body of the constructor is executed in the reverse order to the call order, as the `super()` call can occur only as the first statement in a constructor. This order of execution ensures that the constructor from the `Object` class is completed first, followed by the constructors in the other classes down to the class being instantiated in the inheritance hierarchy. This is called (subclass–superclass) *constructor chaining*. The output from Example 5.7 clearly illustrates this chain of events when an object of the class `TubeLight` is created.

If a constructor at the end of a `this()` chain (which may not be a chain at all if no `this()` call is invoked) does not have an explicit call to `super()`, the call `super()` (without the parameters) is implicitly inserted by the compiler to invoke the no-argument constructor of the superclass. In other words, if a constructor has neither a `this()` call nor a `super()` call as its first statement, the compiler inserts a `super()` call to the no-argument constructor in the superclass. The code

Click here to view code image

```
class A {
  A() {}                // No-argument constructor.
  // ...
}

class B extends A {     // No constructors.
  // ...
}
```

is equivalent to

Click here to view code image

```
class A {
  A() { super(); }    // (1) Call to no-argument superclass constructor insert
  // ...
}

class B extends A {
  B() { super(); }    // (2) Default constructor inserted.
  // ...
}
```

where the compiler inserts a `super()` call in the no-argument constructor for class `A` at (1) and inserts the default constructor for class `B` at (2). The `super()` call at (2) will result in a call to the no-argument constructor in `A` at (1), and the `super()` call at (1) will result in a call to the no-argument constructor in the superclass of `A`—that is, the `Object` class.

If a superclass defines just non-zero argument constructors (i.e., only constructors with parameters), its subclasses cannot rely on the implicit `super()` call being inserted. This will be flagged as a compile-time error. The subclasses must then explicitly call a superclass constructor, using the `super()` construct with the right arguments.

Click here to view code image

```
class NeonLight extends TubeLight {
  // Field
  private String sign;

  NeonLight() {                                // (1)
    super(10, 2, 100, true, "Roof-top");       // (2) Cannot be commented out.
    sign = "All will be revealed!";
  }
  // ...
}
```

The preceding declaration of the subclass `NeonLight` provides a no-argument constructor at (1). The call of the constructor at (2) in the superclass `TubeLight` cannot be omitted. If it is omitted, any insertion of a `super()` call (with no arguments) in this constructor will try to match a no-argument constructor in the superclass `TubeLight`, which provides only non-zero argument constructors. The class `NeonLight` will not compile unless an explicit valid `super()` call is inserted at (2).

If the superclass provides just non-zero argument constructors (i.e., it does not have a no-argument constructor), this has implications for its subclasses. A subclass that relies on its default constructor will fail to compile because the default constructor of the subclass will attempt to call the (nonexistent) no-argument constructor in the superclass. A constructor in a subclass must explicitly use the `super()` call, with the appropriate arguments, to invoke a non-zero argument constructor in the superclass. This call is necessary because the constructor in the subclass cannot rely on an implicit `super()` call to the no-argument constructor in the superclass.

Review Questions

5.1 Given the following class declarations:

Click here to view code image

```
// Classes
class Foo {
  private int i;
  public void f() { /* ... */ }
  public void g() { /* ... */ }
}

class Bar extends Foo {
  public int j;
  public void g() { /* ... */ }
}

public class Main {
  public static void main(String[] args) {
    Foo a = new Bar();
    Bar b = new Bar()
    // (1) INSERT STATEMENT HERE
  }
}
```

Which of the following statements can be inserted at (1) without causing a compile-time error?
Select the two correct answers.

a. `b.f();`
b. `a.j = 5;`
c. `a.g();`
d. `b.i = 3;`

5.2 Given the following code:

Click here to view code image

```
class A {
  void doIt() {}
}

class B extends A {
  void doIt() {}
}
```

```
class C extends B {
  void doIt() {}
  void callUp() {
    // (1) INSERT EXPRESSION HERE
  }
}
```

insert the expression that would call the `doIt()` method in `A`. Select the one correct answer.

a. `doIt();`
b. `super.doIt();`
c. `super.super.doIt();`
d. `this.super.doIt();`
e. `A.this.doIt();`
f. `((A) this).doIt();`
g. It is not possible.

5.3 What would be the result of compiling and running the following program?

Click here to view code image

```
class A {
  int max(int x, int y) { (x>y) ? x : y; }
}

class B extends A {
  int max(int x, int y) { return super.max(y, x) - 10; }
}

class C extends B {
  int max(int x, int y) { return super.max(x+10, y+10); }
}

public class UserClass {
  public static void main(String[] args) {
    B b = new C();
    System.out.println(b.max(13, 29));
  }
}
```

Select the one correct answer.

a. The code will fail to compile.
b. The code will compile, but it will throw an exception at runtime.
c. The code will compile and print `13` at runtime.
d. The code will compile and print `23` at runtime.
e. The code will compile and print `29` at runtime.
f. The code will compile and print `39` at runtime.

5.4 What would be the result of compiling and running the following program?

Click here to view code image

```
class Vehicle {
  static public String getModelName() { return "Volvo"; }
  public long getRegNo() { return 12345; }
}

class Car extends Vehicle {
  static public String getModelName() { return "Toyota"; }
  public long getRegNo() { return 54321; }
}

public class TakeARide {
  public static void main(String[] args) {
    Car c = new Car();
    Vehicle v = c;

    System.out.println("|" + v.getModelName() + "|" + c.getModelName() +
                       "|" + v.getRegNo()     + "|" + c.getRegNo() + "|"
  }
}
```

Select the one correct answer.

a. The code will fail to compile.

b. The code will compile and print `|Toyota|Volvo|12345|54321|` at runtime.

c. The code will compile and print `|Volvo|Toyota|12345|54321|` at runtime.

d. The code will compile and print `|Toyota|Toyota|12345|12345|` at runtime.

e. The code will compile and print `|Volvo|Volvo|12345|54321|` at runtime.

f. The code will compile and print `|Toyota|Toyota|12345|12345|` at runtime.

g. The code will compile and print `|Volvo|Toyota|54321|54321|` at runtime.

5.5 Which constructors can be inserted at (1) in `MySub` without causing a compile-time error?

Click here to view code image

```
class MySuper {
  int number;
  MySuper(int i) { number = i; }
}

class MySub extends MySuper {
  int count;
  MySub(int count, int num) {
    super(num);
    this.count = count;
  }

  // (1) INSERT CONSTRUCTOR HERE
}
```

Select the one correct answer.

a. `MySub() {}`

b. `MySub(int count) { this.count = count; }`

c. `MySub(int count) { super(); this.count = count; }`

d. `MySub(int count) { this.count = count; super(count); }`

e. `MySub(int count) { this(count, count); }`

f. `MySub(int count) { super(count); this(count, 0); }`

5.6 Which of the following statements is true?

Select the one correct answer.

a. A `super()` or `this()` call must always be provided explicitly as the first statement in the body of a constructor.

b. If both a subclass and its superclass do not have any declared constructors, the implicit default constructor of the subclass will call `super()` when run.

c. If neither `super()` nor `this()` is specified as the first statement in the body of a constructor, `this()` will implicitly be inserted as the first statement.

d. If `super()` is the first statement in the body of a constructor, `this()` can be declared as the second statement.

e. Calling `super()` as the first statement in the body of a constructor of a subclass will always work, since all superclasses have a default constructor.

5.7 What will the following program print when run?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    B b = new B("Test");
  }
}

class A {
  A() { this("1", "2"); }

  A(String s, String t) { this(s + t); }

  A(String s) { System.out.println(s); }
}

class B extends A {
  B(String s) { System.out.println(s); }

  B(String s, String t) { this(t + s + "3"); }

  B() { super("4"); };
}
```

Select the one correct answer.

a. It will just print `Test`.

b. It will print `Test` followed by `Test`.

c. It will print `123` followed by `Test`.

d. It will print `12` followed by `Test`.

e. It will print `4` followed by `Test`.

5.4 Abstract Classes and Methods

The keyword `abstract` is used in the following contexts in Java:

- Declaring `abstract` classes
- Declaring `abstract` methods in classes (p. 224), in interfaces (p. 240), and in enum types (p. 294)

Abstract Classes

A *concrete* class is one that defines, by virtue of its `public` methods, a *contract* for services it guarantees its clients and provides the *implementation for all the methods* necessary to fulfill that contract. Clients can readily instantiate a concrete class and use its objects.

In certain cases, a class might want to define the contract for the services, but only provide *partial implementation* for its contract. Such a design decision might be necessary if the abstraction the class represents is so general that certain aspects need to be specialized by subclasses to be of practical use, but at the same time guarantee that these *will* be implemented by the subclasses. This design strategy can be implemented by using *abstract classes*. Clients cannot instantiate an `abstract` class, but now its concrete subclasses must be instantiated to provide the necessary objects.

The class `Vehicle` might be declared as an `abstract` class with a partially implemented contract to represent the general abstraction of a vehicle, as creating instances of the class would not make much sense. Its non-`abstract` subclasses, like `Car` or `Bus`, would then provide the implementation necessary to fulfill the contract of the superclass `Vehicle`, making the abstraction more concrete and useful.

The Java SE Platform API contains many `abstract` classes. The `abstract` class `java.lang.Number` is the superclass of wrapper classes that represent numeric values as objects (§8.3, p. 434). The Java Collections Framework makes heavy use of `abstract` classes in implementing commonly used collection data structures (§15.1, p. 783).

Declaring an `abstract` *Class*

An `abstract` class is declared with the modifier `abstract` in its class header. In Example 5.8, the class `Light` at (1) is declared as an `abstract` class. It also declares an `abstract` method `energyCost()` at (2), which has no method body and is essentially a method header (p. 224).

Click here to view code image

```
abstract class Light {                                          // (1) Abstract cla
  //...
  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)           // (2) Method heade
      throws InvalidHoursException;                             // No method body
}
```

If a class has one or more `abstract` methods, it must be declared as `abstract`, as it is *incomplete*. In Example 5.8, if the `abstract` keyword is omitted from the header of the class `Light` at (1), the compiler will issue an error, as the class declares an `abstract` method and is therefore incomplete.

Like a normal class, an `abstract` class can declare class members, constructors, and initializers. The `abstract` class `Light` in Example 5.8 declares three instance fields— one non-zero argument constructor and three instance methods—in addition to the `abstract` method at (2).

A class that is declared `absract` cannot be instantiated, regardless of whether it has `abstract` methods or not.

Click here to view code image

```
Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time error!
```

The UML class diagram for the inheritance relationship in Example 5.8 is depicted in Figure 5.3. Note that an `abstract` class name and an `abstract` method name are shown in *italics* to distinguish them from concrete classes and methods.

Extending an `abstract` *Class*

A class might choose the design strategy with `abstract` methods to dictate certain behavior, but allow its subclasses the freedom to provide the relevant implementation. An `abstract` class forces its subclasses to provide the subclass-specific functionality stipulated by its `abstract` methods, which is needed to fully implement its abstraction. In other words, subclasses of the `abstract` class have to take a stand and provide implementations of any inherited `abstract` methods before objects can be created. In Example 5.8, since the class `Light` is `abstract`, it forces its *concrete* (i.e., non-`abstract`) subclass to provide an implementation for the `abstract` method `energyCost()`. The concrete subclass `TubeLight` provides an implementation for this method at (3).

Click here to view code image

```
class TubeLight extends Light {
  // ...
  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
```

Creating an object of a subclass results in the fields of all its superclasses, whether these classes are `abstract` or not, to be created and initialized—that is, they *exist* in the subclass object. As with normal classes, the inheritance relationship between classes allows references of the `abstract` superclass type to be declared and used to refer to objects of their subclasses—that is, these references exhibit polymorphic behavior (p. 278).

Figure 5.3 *Class Diagram for Example 5.8*

In Example 5.8, the class `Factory` creates an instance of the subclass `TubeLight` at (6). The `private` fields declared in the `abstract` superclass `Light` can be accessed indirectly by invoking the `public` methods it provides on objects of the subclass `TubeLight`. The subclass reference `cellarLight` is used to invoke `public` methods in the superclass in the following code:

Click here to view code image

```
TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6)
cellarLight.switchOff();                             // Method in superclass
System.out.println(cellarLight.isOn());              // Method in superclass: false
```

The subclass reference `cellarLight` of course can be used to invoke subclass-specific methods, as shown at (7).

Click here to view code image

```
System.out.printf("Energy cost ($): %2.2f%n",
    cellarLight.energyCost(40));                     // (7) Using subclass reference
```

References of an `abstract` superclass can be declared and assigned reference values of its subclass objects, as shown at (8). Superclass references can be used to manipulate subclass objects, as shown at (9), where the `energyCost()` method from the subclass `TubeLight` is executed.

Click here to view code image

```
Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8)
System.out.printf("Energy cost ($): %2.2f%n",
    nightLight.energyCost(30));             // (9) Using superclass reference
                                            // Invokes method in subclass
                                            // Requires throws clause at (4)
```

Note that using the subclass reference `cellarLight` at (7) to invoke the method `energyCost()` cannot throw a checked exception, as readily seen from its declaration in the subclass `TubeLight`. However, using the superclass reference `nightLight` at (9) to invoke the method `energyCost()` can throw a checked exception, as seen from the method declaration in the superclass `Light`. At compile time, only the static type of the reference is known, namely `Light`, and the method `energyCost()` in this class throws a checked `InvalidHoursException` (§7.2, p. 374). The `throws` clause in the `main()` method at (4) specifies this exception—otherwise, the code will not compile.

In the code below, the class `AbstractArt` at (2) must be declared as `abstract` as it does not implement the `abstract` method `paint()` from its superclass `Art` at (1).

Click here to view code image

```
abstract class Art { abstract void paint(); }            // (1) Abstract class

abstract class AbstractArt extends Art {}                // (2) Must be abstra

class MinimalistArt extends AbstractArt {                // (3) Concrete class
  @Override void paint() { System.out.println(":-)"); }  // (4) Concrete metho
}

abstract class PostModernMinimalistArt
                                 extends MinimalistArt { // (5) Abstract class
  @Override void paint() { System.out.println(":-("); }  // (6) Concrete metho
                                                         //     overrides (4)
}

class ArtsyFartsy extends PostModernMinimalistArt {}     // (7) Concrete class
```

Analogous to a normal class, an `abstract` class can only extend a single non-`final` class that can be either concrete or `abstract`. In the code above, the `abstract` class `AbstractArt` at (2) extends the `abstract` class `Art`, and the `abstract` class

`PostModernMinimalistArt` at (5) extends the concrete class `MinimalistArt`.

A non-`final` concrete class, which by definition has no `abstract` methods, can be *considered incomplete* by declaring it as `abstract`. The `PostModernMinimalistArt` class at (5) is declared `abstract` and considered incomplete, even though it is concrete. It cannot be instantiated. However, its subclass `ArtsyFartsy` at (7) is a concrete class, as it inherits the concrete method `paint()` from its `abstract` superclass `PostModernMinimalistArt`.

A class cannot be declared both `final` and `abstract`—that would be a contradiction in terms: A `final` class cannot be extended, but an `abstract` class is incomplete or considered to be incomplete and must be extended.

An `abstract` class should not be used to implement a class that cannot be instantiated. The recommended practice is to only provide a zero-argument constructor that is `private`, thus making sure that it is never invoked in the class.

In many ways `abstract` classes and interfaces are similar, and interfaces can be used with advantage in many cases. However, if private state should be maintained with instance members, then `abstract` classes are preferred, as interfaces do not have any notion of state.

Analogous to a normal class, an `abstract` class can implement multiple interfaces (p. 240).

Example 5.8 *Using Abstract Classes*

Click here to view code image

```
// File: Factory.java
// Checked exceptions:
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

abstract class Light {                                     // (1) Abstract cla
  // Fields:
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)      // (2) Method heade
      throws InvalidHoursException;                        // No method body
}
//____________________________________________________________________________
class TubeLight extends Light {
  // Instance fields:
  private int tubeLength;                                  // millimeters
  private int tubeDiameter;                                // millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in supercla
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }

  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
//____________________________________________________________________________
public class Factory {
  public static void main(String[] args) throws InvalidHoursException {  // (4
//  Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time er
    TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6
    cellarLight.switchOff();
    System.out.println(cellarLight.isOn());        // false
    System.out.printf("Energy cost ($): %2.2f%n",
        cellarLight.energyCost(40));               // (7) Using subclass refer
    Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8
    System.out.printf("Energy cost ($): %2.2f%n",
        nightLight.energyCost(30));             // (9) Using superclass refere
                                                // Invokes method in subclass
                                                // Requires throws clause in (
  }
}
```

Output from the program:

```
false
Energy cost ($): 6.00
Energy cost ($): 4.50
```

Abstract Methods in Classes

In this subsection we discuss in more detail declaring and overriding `abstract` methods in classes.

Declaring an `abstract` *Method*

An `abstract` method in an `abstract` class has the following syntax:

Click here to view code image

```
access_modifier abstract return_type method_name (formal_parameter_list)
                                                   throws_clause;
```

An `abstract` method does not have an implementation; that is, no method body is defined for an `abstract` method, and only the *method header* is provided in the class declaration. The keyword `abstract` is mandatory in the header of an `abstract` method declared in a class. Its class is then incomplete and must be explicitly declared as `abstract`. Subclasses of an `abstract` class must then override the `abstract` method to provide the method implementation; otherwise, they must also be declared as `abstract`.

Overriding an `abstract` *Method*

When overriding an `abstract` method from the superclass, the notation `@Override` should always be used in the overriding method in the subclass. The compiler will issue an error if the override criteria are not satisfied.

The accessibility of an `abstract` method declared in a top-level class cannot be `private`, as subclasses would not be able to override the method and provide an implementation. Thus an `abstract` method in a top-level class can only have `public`, `protected`, or package accessibility.

In Example 5.8, the `abstract` instance method in the `abstract` superclass `Light` has the following declaration:

Click here to view code image

```
protected abstract double energyCost(int noOfHours)       // (2) Method header
    throws InvalidHoursException;                          // No method body
```

It has `protected` access and has type `double` as the return type. Its method signature is `energyCost(int)`, and it throws the checked `InvalidHoursException`.

The implementation of the `abstract` method in the subclass `TubeLight` has the following declaration:

Click here to view code image

```
@Override public double energyCost(int noOfHours) {       // (3)
  return  0.15 * noOfHours;
}
```

It has `public` access and has type `double` as the return type. Its method signature is `energyCost(int)`, and it has no `throws` clause. Widening the access to `public` access and throwing no checked exceptions are allowed according to the override criteria.

Since an `abstract` method must be overridden to provide an implementation, only an instance method can be declared as `abstract`. Since `static` methods cannot be overridden, declaring an `abstract static` method makes no sense, and the compiler will report an error.

An `abstract` method can be overloaded just like a normal method. The following method declaration in either the superclass or the subclass overloads the method named `energyCost`, as it has a different signature: `energyCost()`.

Click here to view code image

```
public double energyCost() {          // Overloaded
  return 1.75;
}
```

If an attempt to override or overload an `abstract` method fails, the compiler will issue an error. If either of these two methods is declared in a subclass of the `Light` class, the compiler will issue an error.

Click here to view code image

```
@Override
double energyCost(int numOfHours) {   // Not overridden! Narrows accessibility
  return 2.0 * numOfHours;
}

public Double energyCost(int numOfHours) {   // Not overloaded! Duplicate metho
  return 3.5 * numOfHours;
}
```

An `abstract` method or a non-`final` concrete method in a class can be overridden by an `abstract` method in a subclass. This is governed by the same rules for method overriding.

A method cannot be both `final` and `abstract`—that would be a contradiction in terms: A `final` method cannot be overridden, but an `abstract` method must be overridden to provide an implementation.

For a discussion of `abstract` methods in top-level interfaces, see §5.6, p. 240. Abstract methods can also be declared in an enum type, if the enum type contains constant-specific class bodies that implement these methods (p. 294).

5.5 Final Declarations

The keyword `final` can be used in the following contexts in Java:

- Declaring `final` classes (p. 225)
- Declaring `final` members in classes (p. 226), and in enum types (p. 294)
- Declaring `final` local variables in methods and blocks (p. 231)
- Declaring `final static` variables in interfaces (p. 254)

Final Classes

A class can be declared `final` to indicate that it cannot be extended—that is, one cannot define subclasses of a `final` class. In other words, the class behavior cannot be changed by extending the class. This implies that one cannot override or hide any methods declared in such a class. Its methods behave as `final` methods (p. 226). A `final` class marks the lower boundary of its *implementation inheritance hierarchy*.

A *concrete* class is a class that has only *concrete* methods—that is, methods that are non-`abstract`, and therefore have an implementation. Only a concrete class can be declared `final`. If it is decided that the class `TubeLight` at (12) in Example 5.9 may not be extended, it can be declared `final`. Any attempt to specify the class name `TubeLight` in an `extends` clause will result in a compile-time error.

Click here to view code image

```
final class TubeLight extends Light {                    // (12)
  // ...
}

class NeonLight extends TubeLight {                      // Compile-time error!
  // ...
}
```

A `final` class must be complete, whereas an `abstract` class is considered incomplete. Classes, therefore, cannot be both `final` and `abstract` at the same time. Interfaces are implicitly `abstract`, and therefore cannot be declared as `final`. A `final` class and an interface with only `abstract` methods represent two diametrical strategies when it comes to providing an implementation: the former with all the methods implemented and the latter with none. An `abstract` class or an interface with partial implementation represents a compromise between these two strategies.

The Java SE Platform API includes many `final` classes—for example, the `java.lang.String` class and the wrapper classes for primitive values.

Final Methods in Classes

A `final` method in a class is a *concrete* method (i.e., has an implementation) and cannot be overridden or hidden in any subclass. Any normal class can declare a `final` method. The class need not be declared `final`.

In Example 5.9, the non-`final` class `Light` defines the `final` method `setWatts()` at (10).

Click here to view code image

```
class Light {                                    // (1)
  // ...
  public final void setWatts(int watt) {         // (10) Final instance method
    noOfWatts = watt;
  }
}
```

The subclass `TubeLight` attempts to override the `final` method `setWatts()` from the superclass `Light` at (14), which is not permitted.

Click here to view code image

```
class TubeLight extends Light {                  // (12)
  // ...
  @Override
  public void setWatts(int watt) {   // (14) Cannot override final method at (
    noOfWatts = 2*watt;
  }
}
```

A call to a `final` method is bound at compile time; as such, a method cannot be overridden by a subclass and therefore there is no dynamic lookup to perform at runtime. A method that is declared `final` allows the compiler to do code optimization by replacing its method call with the code of its method body, a technique called *inlining*. Use of the `final` keyword for exceptional performance enhancement is not recommended, as the compiler is too smart to fall for that trick.

A subclass cannot override or hide a `private` method from its superclass. However, it can define a new method using the same method signature as the `private` method, without regard to the other criteria for overriding: the return type and the `throws` clause. Defining a new method using the same method signature is *not* possible with a `final` method which cannot be overridden or hidden. Declaring a `private` method as `final` in addition is redundant.

Although a constructor cannot be overridden, it also *cannot* be declared `final`.

The `java.lang.Object` class—the superclass of all classes—defines a number of `final` methods (`notify()`, `notifyAll()`, and `wait()`—mostly to synchronize the running of threads) that all objects inherit, but cannot override.

Example 5.9 *Using the* `final` *Modifier*

Click here to view code image

```
import java.util.Arrays;
import static java.lang.System.out;

class Light {                                     // (1)
  // Static final variables:
  public static final double KWH_PRICE = 3.25;    // (2) Constant static variab
  public static final String MANUFACTURER;        // (3) Blank final static fie

  static {                                        // Static initializer block
    MANUFACTURER = "Ozam";                        // (4) Initializes (3)
  }

  // Instance variables:
  int noOfWatts;                                  // (5)
  final String color;                             // (6) Blank final instance f
  final String energyRating;                      // (7) Blank final instance f

  {                                               // Instance initializer block
    color = "off white";                          // (8) Initializes (6)
  }

  // Constructor:
  Light() {
    energyRating = "A++";                         // (9) Initializes (7)
  }

  public final void setWatts(int watt) {          // (10) Final instance method
    this.noOfWatts = watt;
  }

  public static void setKWH(double rate) {
//  KWH_PRICE = rate;                             // (11) Not OK. Final field.
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {                   // (12)
```

```
  static StringBuilder color = new StringBuilder("green");      // (13) Hiding

//@Override
//public void setWatts(int watt) {   // (14) Cannot override final method at (
//  noOfWatts = 2*watt;
//}
}
//____________________________________________________________________________
public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variabl
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final refere

    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
//  carLight.setWatts(10);               // (23) Not OK. Not initialized.

    out.println("Accessing final static fields in class Light:");
    out.println("KWH_PRICE:    " + Light.KWH_PRICE);
    out.println("MANUFACTURER: " + Light.MANUFACTURER);

    out.println("Accessing final instance fields in an object of class Light:"
    out.println("noOfWatts:    " + workLight.noOfWatts);
    out.println("color:        " + workLight.color);
    out.println("energyRating: " + workLight.energyRating);

    out.println("Fun with final parameter args:");
    out.println(Arrays.toString(args));  // Print array.
    out.println("args length: " + args.length);
    args[0] = "1";                       // (24) OK. Modifying array state.
    out.println(Arrays.toString(args));  // Print array.
//  args = null;                         // (25) Not OK. Final parameter.
//  args.length = 10;                    // (26) Not OK. Final instance field.
  }
}
```

Output from the program when run with the following command:

Click here to view code image

```
>java Warehouse One Two Three
Accessing final static fields in class Light:
KWH_PRICE:    3.25
MANUFACTURER: Ozam
Accessing final instance fields in an object of class Light:
noOfWatts:    100
color:        off white
energyRating: A++
Fun with final parameter args:
[One, Two, Three]
args length: 3
[1, Two, Three]
```

Final Variables

A `final` *variable* is a variable whose value cannot be changed during its lifetime once it has been initialized. A `final` variable is declared with the keyword `final` in its declaration. Any attempt to reassign a value will result in a compile-time error.

Declaring a variable as `final` has the following implications:

- A `final` variable of a primitive data type cannot change its value once it has been initialized.
- A `final` variable of a reference type cannot change its reference value once it has been initialized. This effectively means that a `final` reference will always refer to the same object. However, the keyword `final` has no bearing on whether the *state of the object* denoted by the reference can be changed.
- The compiler may perform code optimizations for `final` variables, as certain assumptions can be made about such code.

For all paths of execution through the code, the compiler checks that a `final` variable is assigned only once before it is accessed, and when in doubt, the compiler issues an error (p. 232).

A *blank* `final` *variable* is a `final` variable whose declaration does not specify an initializer expression.

A *constant variable* is a `final` variable of either a primitive type or type `String` that is initialized with a *constant expression*. Constant variables can be used as `case` labels of a `switch` statement.

We distinguish between two kinds of `final` variables: `final` *fields* that are members in a class declaration and `final` *local variables* that can be declared in methods and blocks. A `final` *field* is either `static` or non-`static` depending on whether it is declared as a `static` or non-`static` member in the class.

Example 5.9 provides examples of different kinds of `final` variables. The declarations from Example 5.9 are shown here.

Click here to view code image

```
class Light {                                    // (1)
  public static final double KWH_PRICE = 3.25;   // (2) Constant static variab
  public static final String MANUFACTURER;       // (3) Blank final static fie

  int noOfWatts;                                 // (5) Non-final instance fie
  final String color;                            // (6) Blank final instance f
  final String energyRating;                     // (7) Blank final instance f
  // ...
}
```

```
public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variabl

    final Light alarmLight;              // (20) Blank final local variable.

    Light carLight;                      // (22) Non-final local variable.
    // ...
  }
}
```

In the next two subsections we elaborate on `final` fields in classes and `final` local variables that can be declared in methods and blocks.

Final Fields in Classes

A `final` field must be explicitly initialized only once with an initializer expression, either in its declaration or in an initializer block. A `final` instance field can also be initialized in a constructor. After the class completes initialization, the `final static` fields of the class are all guaranteed to be initialized. After a constructor completes execution, the `final` instance fields of the current object are all guaranteed to be initialized. The compiler ensures that the class provides the appropriate code to initialize the `final` fields.

In Example 5.9, the class `Light` defines two `final static` fields at (2) and (3). The field `KWH_PRICE` is a constant variable that is initialized by the constant expression `3.25` in the declaration at (2). The field `MANUFACTURER` is a blank `final static` field that is initialized in the `static` initializer block at (4). All `static` initializer blocks are executed at class initialization. Note that a blank `final static` field *must* be initialized in a `static` initializer block; otherwise, the compiler will issue an error. An attempt to change the value of the `final static` field `KWH_PRICE` at (11) in a method results in a compile-time error.

In Example 5.9, the class `Light` also defines two `final` instance fields at (6) and (7). The instance field `color` is a blank `final` instance field that is initialized in the instance initializer block at (8), and the instance field `energyRating` is a blank `final` instance field that is initialized in the constructor at (9). If either the assignment at (8) in the instance initializer block or the assignment at (9) in the constructor is removed, it will result in a compile-time error.

Note that a blank `final` instance field *must* be initialized either in an instance initializer block or in a constructor; otherwise, the code will not compile. Each time an object is created using the `new` operator with a constructor call, all instance initializer blocks are executed, but not necessarily all constructors, as this depends on constructor chaining. If the class has several constructors, code must make sure that a blank `final` instance field is initialized no matter which constructor is called to initialize the object.

Since `final static` fields are initialized at class initialization time and `final` instance fields are initialized at object creation time, the compiler will issue an error if an attempt is made to assign a value to a `final` field inside any method.

Analogous to a non-`final` field, a `final` field with the same name *can* be hidden in a subclass—in contrast to `final` methods that can neither be overridden nor hidden. In Example 5.9, the blank `final` instance field `color` of type `String` at (6) in class `Light` is hidden by the `static` field `color` of type `StringBuilder` at (13) in subclass `TubeLight`.

Fields that are `final` and `static` are commonly used to define *manifest constants* (also called *named constants*). For example, the minimum and maximum values of the numerical primitive types are defined by `final static` fields in the respective wrapper classes. The `final static` field `Integer.MAX_VALUE` defines the maximum `int` value. The field `java.lang.System.out` is a `final static` field. Fields defined in an interface are implicitly `final static` fields (p. 254), as are the enum constants of an enum type (p. 287).

Final Local Variables in Methods

A `final` local variable need not be initialized in its declaration—that is, it can be a blank `final` local variable—but it must be initialized in the code before it is accessed. However, the compiler does not complain as long as the local variable is not accessed—this is in contrast to `final` fields that must be explicitly assigned a value, whether they are accessed or not.

In Example 5.9, the `main()` method at (15) in the class `Warehouse` defines a `final` local reference `workLight` at (16). The state of the object denoted by the reference `workLight` is changed at (17), but an attempt to change the value of the `final` instance field `color` of this object at (18) does not succeed. The compiler also reports an error at (19), since the reference value of the `final` local variable `work-Light` cannot be changed. A blank `final` local reference `alarmLight` is declared at (20). The compiler reports an error when an attempt is made to use this reference at (21) before it is initialized. The non-`final` local reference `carLight` at (22) also cannot be accessed at (22) before it is initialized. However, note that the compiler does not complain as long as the local variables at (20) and (22) are not accessed.

Click here to view code image

```
     final Light workLight = new Light(); // (16) Final local variable.
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final refere

    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
//  carLight.setWatts(10);               // (23) Not OK. Not initialized.
```

Local variables in certain contexts are considered to be *implicitly declared* `final`: an exception parameter of a multi-`catch` clause (§7.6, p. 397) and a local variable that refers to a resource in a `try`-with-resources statement (§7.7, p. 407).

`final` Parameters

A formal parameter can be declared with the keyword `final` preceding the parameter declaration in the method header. A `final` parameter is a special case of a blank `final` local variable. It is initialized with the value of the corresponding argument in the method call, before the code in the body of the method is executed. Declaring parameters as `final` prevents their values from being changed inadvertently in the method. A formal parameter's declaration as `final` does not affect the caller's code.

In the declaration of the `main()` method at (15) in Example 5.9, the parameter `args` is declared as `final`. The array `args` will be initialized with the program arguments on the command line when execution starts.

Click here to view code image

```
public static void main(final String[] args) {  // (15) Final parameter
  // ...
  out.println("Fun with final parameter args:");
  out.println(Arrays.toString(args));         // (24) Print array.
  out.println("args length: " + args.length); // (25) Access final field
  args[0] = "1";                              // (26) OK. Modifying array stat
  out.println(Arrays.toString(args));         // Print array.
//args = null;                                // (27) Not OK. Final parameter.
```

```
  //args.length = 10;                         // (28) Not OK. Final instance fi
  }
```

The state of the array object `args` is printed at (24). Its `final` field `length` is accessed at (25). The state of the `args` array is changed at (26), and is printed again. However, the compiler will not allow the reference value of the `final` parameter `args` to be changed at (27), nor will it allow the value of the `final` instance field `args.length` to be changed at (28).

Definite Assignment Analysis for Final Variables

The *name* of a `final` variable can occur in the following contexts: in its declaration when it is declared, in the context of an assignment when it is assigned a value, and in the context where its value is accessed in an expression.

The analysis performed by the compiler determines whether the blank `final` variable is initialized *before* its value is accessed. This involves checking whether a blank `final` variable has been assigned a value on any possible path of execution to where the value of the variable is accessed. In technical terms, this means a blank `final` variable must be *definitely assigned* before any access.

Click here to view code image

```
  final int k;                // Declaration: blank final local variable
  k = 10;                     // (1) Assignment
  System.out.println(k);      // (2) Access: k is definitely assigned.

  boolean status = true;      // Non-constant variable.
  final int j;                // Declaration: blank final local variable
  if (status) {               // (3) Value of conditional expression not evaluated
    j = 5;                    // (4) Conditional assignment
  }
  System.out.println(j);      // (5) Access: j is not definitely assigned!
```

The blank `final` local variable `k` is accessed at (2) after it has been assigned a value at (1). It is *definitely assigned* before access at (2). The situation is different at (5). The compiler can deduce that the blank `final` local variable `j` can be assigned a value in the `if` block, depending on the conditional expression in the `if` statement. In other words, the structure of the `if` statement is considered in the analysis, but not the non-constant conditional expression. The compiler cannot guarantee that the assignment at (4) will be executed. The conclusion being that the blank `final` local variable `j` is *not definitely assigned* before access at (5). The statement at (5) does not compile. However, if the conditional expression in the `if` statement at (3) is replaced with the `boolean` literal `true`, the compiler can determine that the assignment at (4) will be executed and the variable `j` at (5) becomes definitely assigned before access.

The flow analysis performed is conservative. The compiler does not always evaluate all expressions, particularly method calls. However, it does evaluate `boolean`-valued constant expressions, and it also takes into consideration the syntax of statements and expressions, including the use of the conditional operators `!`, `&&`, `||`, and `? :`.

For each assignment to a blank `final` variable, the compiler must also determine that it is assigned *exactly once*—in other words, it has not already been assigned a value. This means that, given an assignment to a blank `final` local variable, there should not be any other assignment to this variable on any possible path of execution prior to this assignment. In technical terms, this means a blank `final` variable must be *definitely unassigned* before any assignment.

Click here to view code image

```
int n = 5;
final int k;            // Declaration: blank final local variable
if (n >= 4) {           // (1) Value of conditional expression not evaluated
  k = 6;                // (2) Conditional assignment: k is definitely unassig
}
k = 12;                 // (3) Assignment: k is not definitely unassigned!
System.out.println(k);  // (4) Access: k is definitely assigned
```

The blank `final` local variable `k` in the assignment at (2) is definitely unassigned, as there is no other assignment on a path of execution before this assignment. However, the variable `k` at (3) is *not* definitely unassigned, as the assignment at (2) in the `if` statement can be executed on a path of execution before the assignment at (3). The compiler reports that the variable `k` at (3) may already have been assigned a value. The assignment at (3) does not compile. Note that the variable `k` at (4) is definitely assigned before the access, because of the assignment at (3). Removing the assignment statement at (3) makes the variable `k` at (4) *not* definitely assigned, as in the previous example.

If the code is modified by replacing the `if` statement at (3) with an `if-else` statement, where the assignment at (3) is done in the `else` block, the variable k at (3) also becomes definitely unassigned—there is no assignment to the variable `k` before this assignment. Regardless of which assignment executes in the `if-else` statement, the variable `k` at (4) is definitely assigned before access.

Click here to view code image

```
int n = 5;
final int k;            // Declaration: blank final local variable
if (n >= 4) {           // (1) Value of conditional expression not evaluated
  k = 6;                // (2) Conditional assignment: k is definitely unassig
} else {
  k = 12;               // (3) Conditional assignment: k is definitely unassig
}
System.out.println(k);  // (4) Access: k is definitely assigned
```

The two rules of definitely assigned before an access and definitely unassigned before an assignment help to determine correct usage of blank `final` variables. For non-`final` *local variables*, the rule for definitely assigned before an access is sufficient, as the value of such variables can be changed.

The examples below shed more light on the usage of `final` variables.

Click here to view code image

```
{
  final int i = 10;
  i++;                 // Not OK. Side effect changes the value of i.
}

{
  for (int i = 0; i < 10; i++) {
    final int j = i; // OK. Final variable goes out of scope after each iterat
  }
}

{
  final int j;
  for (int i = 0; i < 10; i++) {
    j = i;   // Not OK. Loop either not executed and j not initialized
             // or each iteration will assign a new value to j.
  }
  System.out.println(j); // Not OK. Not guaranteed that j is initialized.
}
```

Review Questions

5.8 Which one of the following class declarations is a valid declaration of a class that cannot be instantiated?

Select the one correct answer.

a. Click here to view code image

```
class Ghost          { abstract void haunt(); }
```

b. Click here to view code image

```
abstract class Ghost { void haunt(); }
```

c. Click here to view code image

```
abstract class Ghost { void haunt() {}; }
```

d. Click here to view code image

```
abstract Ghost       { abstract void haunt(); }
```

e. Click here to view code image

```
abstract class Ghost { abstract haunt(); }
```

5.9 Given the following classes:

Click here to view code image

```
public class Animal {
  // (1)
}

public abstract class Cat extends Animal {
  // (2)
}
```

and the following method definition:

Click here to view code image

```
public abstract void eat();
```

where can this abstract method be inserted?
Select the one correct answer.

a. It can be inserted at (1).
b. It can be inserted at (2).
c. It can be inserted at both (1) and (2).
d. It cannot be inserted into any of the classes.

5.10 Which statement is true about an `abstract` class?

Select the one correct answer.

a. An `abstract` class cannot have a constructor.
b. An `abstract` class cannot have concrete methods.
c. An `abstract` class cannot override methods from its supertypes.
d. An `abstract` class cannot be instantiated with the `new` operator.

5.11 Which statement is true about the following classes?

Click here to view code image

```
abstract class LivingOrganism {
  public abstract void feed();
}

class Bacteria extends LivingOrganism {
  @Override
  public void feed()  {  }
  public void feed(int quantity)  {  }          // (1)
}

public class Lab {
  public static void main(String[] args) {
    LivingOrganism org = new Bacteria();        // (2)
    org.feed();
    ((Bacteria)org).feed(10);                   // (3)
  }
}
```

Select the one correct answer.

a. Class `Bacteria` cannot overload the `feed()` method at (1).

b. An `@Overload` annotation can be specified for the `feed()` method at (1).

c. An `@Override` annotation can be specified for the `feed()` method at (1).

d. An instance of `Bacteria` can be assigned to the `org` variable at (2).

e. Class `Lab` at (3) cannot invoke the `feed()` method at (1).

5.12 Which of the following statements are true? Select the two correct answers.

a. In Java, the `extends` clause is used to specify the inheritance relationship.

b. The subclass of a non-`abstract` class can be declared as `abstract`.

c. All members of the superclass are inherited by the subclass.

d. A `final` class can be `abstract`.

e. A class in which all the members are declared `private` cannot be declared as `public`.

5.13 Which one of the following class declarations is a valid declaration of a class that cannot be extended?

Select the one correct answer.

a. `class Link { }`

b. `abstract class Link { }`

c. `final class Link { }`

d. `abstract final class Link { }`

5.14 Given the following source code, which comment line can be uncommented without introducing errors?

Click here to view code image

```
abstract class MyClass {
  abstract void f();
  final    void g() {}
//final    void h() {}                         // (1)

  protected static int i;
  private          int j;
}

final class MyOtherClass extends MyClass {
//MyOtherClass(int n) { m = n; }               // (2)

  public static void main(String[] args) {
    MyClass mc = new MyOtherClass();
  }

  void f() {}
  void h() {}
//void k() { i++; }                            // (3)
//void l() { j++; }                            // (4)

  int m;
}
```

Select the one correct answer.

- **a.** (1)
- **b.** (2)
- **c.** (3)
- **d.** (4)

5.15 Which of the following statements are true about modifiers? Select the two correct answers.

- **a.** Abstract classes can declare `final` methods.
- **b.** Fields can be declared as `abstract`.
- **c.** Non-`abstract` methods can be declared in `abstract` classes.
- **d.** Abstract classes can declare `default` methods.
- **e.** Abstract classes can be declared as `final`.

5.16 Which statement is true about the following classes?

Click here to view code image

```
public abstract class Animal {
  public static final MAX_SIZE = 10;
  public abstract void measure(int size);
}

public class Cat extends Animal {
  private int size;
  @Override
  public final void measure(int size) {
    this.size = (size < Animal.MAX_SIZE) ? size : Animal.MAX_SIZE;
  }
}
```

Select the one correct answer.

a. The compilation fails because of the `final` variable declaration in the abstract class.

b. The compilation fails because of the `final` method overriding an `abstract` method.

c. The compilation fails because `@Override` annotation cannot be applied to a `final` method.

d. The compilation succeeds.

5.17 Which statement is true about `final` classes and methods?

Select the one correct answer.

a. A `final` class cannot contain `abstract` methods.

b. An `abstract` class cannot contain `final` methods.

c. A `final` class cannot contain `final` methods.

d. A `final` method cannot override an `abstract` method.

5.6 Interfaces

An interface defines a *contract* for services that classes can implement. Objects of such classes guarantee that this contract will be honored.

Before diving into interfaces, an overview of the inheritance relationship between *classes* can be useful. The `extends` clause in a class definition only allows *linear inheritance* between classes—that is, a subclass can only extend one superclass. A superclass reference can refer to objects of its own type and of its subclasses strictly according to the linear inheritance hierarchy. Note that this inheritance relationship between classes comprises both *inheritance of type* (i.e., a subclass inherits the type of its superclass and can act as such) and *inheritance of implementation* (i.e., a subclass inherits methods and fields from its superclass). Since this relationship is linear, it rules out *multiple inheritance of implementation*, in which a subclass can inherit implementation directly from more than one direct superclass.

As we shall see in this section, *interfaces* not only allow new named reference types to be introduced, but their usage can result in both *multiple inheritance of type* and *multiple inheritance of implementation*. As we shall also see, multiple inheritance of type does not pose any problems, but multiple inheritance of implementation does and is disallowed by the compiler.

Defining Interfaces

A top-level interface has the following syntax:

Click here to view code image

```
access_modifier interface interface_name
                            optional_type_parameter_list
                            optional_extends_interface_clause // Interface header
{ // Interface body
     abstract_method_declarations
     default_method_declarations
     static_method_declarations
     private_instance_method_declarations
     private_static_method_declarations
     constant_declarations
     member_type_declarations
}
```

In the interface header, the name of the interface is preceded by the keyword `interface`. In addition, the interface header can specify the following information:

- The *access modifier* can be `public` or `private`. Lack of an access modifier implies package accessibility.
- The *optional type parameter list* specifies a comma-separated list of any formal type parameters enclosed by angle brackets (`<>`) for declaring a *generic interface* (§11.2, p. 572).
- The *optional extends interface clause* specifies a comma-separated list of any superinterfaces that the interface extends (p. 244).

The interface body can contain *member declarations* that include any of the following:

- *Abstract method declarations* (p. 240)
- *Default method declarations* (p. 246)
- *Static method declarations* (p. 251)
- *Private instance and static method declarations* (p. 252)
- *Constant declarations* (p. 254)
- *Member type declarations* (§9.1, p. 491)

An interface is `abstract` by definition, which means that it cannot be instantiated. Declaring an interface as `abstract` is superfluous and seldom done in practice. It is the only non-access modifier that can be specified for a top-level interface.

The member declarations can appear in any order in the interface body, which can be empty. Since interfaces are meant to be implemented by classes, interface members implicitly have `public` access and the `public` modifier can be omitted. The following declaration is an example of a bare-bones interface that has an empty body:

```
interface Playable { }
```

Interfaces with empty bodies can be used as *markers* to *tag* classes as having a certain property or behavior. Such interfaces are also called *ability* interfaces. The Java SE Platform API provides several examples of such marker interfaces—for example, `java.lang.Cloneable`, `java.io.Serializable` (§20.5, p. 1261), and `java.util.EventListener`. However, *annotations* (Chapter 25, p. 1555) are a better solution than marker interfaces for attaching metadata to class definitions.

Table 5.2 and Table 5.3 summarize the salient properties of member declarations that can be included in an interface, and which we will elaborate on in this section.

Table 5.2 *Summary of Member Declarations in an Interface (Part I)*

[illegible]					
Access modifier: [illegible]	[illegible]				
Non-access modifier: [illegible]	[illegible]				
Implemented: [illegible]					
Can be overridden: [illegible]					

Table 5.3 *Summary of Member Declarations in an Interface (Part II)*

[illegible]		
Access modifier: [illegible]		
Non-access modifier: [illegible]	`final`	
Implemented: [illegible]		
[illegible]		

Abstract Methods in Interfaces

An interface defines a *contract* by specifying a set of `abstract` and `default` method declarations, but provides implementations only for the `default` methods—not for the `abstract` methods. The `abstract` methods in an interface are all implicitly `abstract` and `public` by virtue of their definitions. Only the modifiers `abstract` and `public` are allowed, but these are invariably omitted. An `abstract` method declaration has the following simple form in a top-level interface:

Click here to view code image

```
return_type method_name (formal_parameter_list) throws_clause;
```

An `abstract` method declaration is essentially a method header terminated by a semicolon (`;`). Note that an `abstract` method is an *instance method* whose implementation will be provided by a class that implements the interface in which the `abstract` method is declared. The `throws` *clause* is discussed in §7.5, p. 388.

The interface `Playable` shown below declares an `abstract` method `play()`. This method is implicitly declared to be `public` and `abstract`. The interface `Playable` defines a contract: To be `Playable`, an object must implement the method `play()`.

Click here to view code image

```
interface Playable {
  void play();              // Abstract method: no implementation
}
```

An interface that has no direct superinterfaces implicitly includes a `public abstract` method declaration for each `public` instance method from the `java.lang.Object` class (e.g., `equals()`, `toString()`, `hashCode()`). These methods are *not* inherited from the `java.lang.Object` class, as only `abstract` method declarations are included in the interface. Their inclusion allows these methods to be called using an interface reference, and their implementation is always guaranteed at runtime as they are either inherited or overridden by all classes.

In contrast to the syntax of `abstract` methods in top-level interfaces, `abstract` methods in top-level classes must be explicitly specified with the keyword `abstract`, and can have `public`, `protected`, or package accessibility.

Functional interfaces, meaning interfaces with a single `abstract` method, are discussed together with lambda expressions in §13.1, p. 675.

The rest of this chapter provides numerous examples of declaring, implementing, and using interfaces.

Implementing Interfaces

A class can implement, wholly or partially, any number of interfaces. A class specifies the interfaces it implements as a comma-separated list of unique interface names in an `implements` clause in the class header. Implementing an interface essentially means that the class must provide implementation for the `abstract` methods declared in the interface. In fact, an `abstract` method declaration is overridden when an implementation is provided by a class. Optionally, the class can also override any `default` methods if necessary.

The criteria for overriding methods in classes also apply when implementing `abstract` and any `default` methods from interfaces. A class must provide a method implementation with the same method signature and (covariant) return type as the declaration in the interface, and it can neither narrow the `public` access of the method nor specify new exceptions in the method's `throws` clause, as attempting to do so would amount to altering the interface's contract, which is illegal.

Best practice advocates using the `@Override` annotation on the implementations of `abstract` and `default` methods (Example 5.10). The compiler checks that such a method satisfies the criteria for overriding another method. This check ensures that the method is not inadvertently overloaded by catching the error at compile time. The annotations also aid in the readability of the code, making obvious which methods are overridden.

It is not enough for a class to provide implementations of methods declared in an interface. The class must also specify the interface name in its `implements` clause in order to reap the benefits of interfaces.

In Example 5.10, the class `Stack` at (2) implements the interface `IStack`. It both specifies the interface name using the `implements` clause in its class header at (2) and provides the implementation for the `abstract` methods in the interface at (3) and (4). Changing the `public` access of these methods in the class will result in a compile-time error, as this would narrow their accessibility.

Example 5.10 *Implementing Interfaces*

Click here to view code image

```
// File: StackUser.java
interface IStack {                                                        // (
  void   push(Object item);
  Object pop();
```

```
}
//_____________________________________________________________________________
class Stack implements IStack {                                          // (
  protected Object[] elements;
  protected int      tos;  // top of stack

  public Stack(int capacity) {
    elements = new Object[capacity];
    tos        = -1;
  }

  @Override
  public void push(Object item) { elements[++tos] = item; }              // (

  @Override
  public Object pop() {                                                  // (
    Object objRef = elements[tos];
    elements[tos] = null;
    tos--;
    return objRef;
  }
  public Object peek() { return elements[tos]; }
}
//_____________________________________________________________________________
interface ISafeStack extends IStack {                                    // (
  boolean isEmpty();
  boolean isFull();
}
//_____________________________________________________________________________
class SafeStack extends Stack implements ISafeStack {                    // (

  public SafeStack(int capacity) { super(capacity); }
  @Override public boolean isEmpty() { return tos < 0; }                 // (
  @Override public boolean isFull()  { return tos >= elements.length-1; } // (
}
//_____________________________________________________________________________
public class StackUser {

  public static void main(String[] args) {                               // (
    SafeStack  safeStackRef  = new SafeStack(10);
    Stack      stackRef      = safeStackRef;
    ISafeStack isafeStackRef = safeStackRef;
    IStack     istackRef     = safeStackRef;
    Object     objRef        = safeStackRef;

    safeStackRef.push("Dollars");                                        // (
    stackRef.push("Kroner");
    System.out.println(isafeStackRef.pop());
    System.out.println(istackRef.pop());
    System.out.println(objRef.getClass());
  }
}
```

Output from the program:

```
Kroner
Dollars
class SafeStack
```

A class can choose to implement only some of the `abstract` methods of its interfaces (i.e., give a partial implementation of its interfaces). The class must then be declared as `abstract`. Note that `abstract` methods cannot be declared as `static`, because they comprise the contract fulfilled by the *objects* of the class implementing the interface. Abstract methods are always implemented as instance methods.

The interfaces that a class implements and the classes that it extends (directly or indirectly) are called *supertypes* of the class. Conversely, the class is a *subtype* of its supertypes. A class can now inherit from multiple interfaces. Even so, regardless of how many interfaces a class implements directly or indirectly, it provides just a *single* implementation of any `abstract` method declared in multiple interfaces.

Single implementation of an `abstract` method is illustrated by the following code, where the `Worker` class at (5) provides only one implementation of the `doIt()` method that is declared in both interfaces, at (1) and (2). The class `Worker` fulfills the contract for both interfaces, as the `doIt()` method declarations at (1) and (2) have the same method signature and return type. However, the class `Combined` at (3) declares that it implements the two interfaces, but does not provide any implementation of the `doIt()` method; consequently, it must be declared as `abstract`.

Click here to view code image

```
interface IA { int doIt(); }                         // (1)

interface IB { int doIt(); }                         // (2)

abstract class Combined implements IA, IB { }        // (3)

public class Worker implements IA, IB {              // (4)
  @Override
  public int doIt() { return 0; }                    // (5)
}
```

Multiple declarations of the same `abstract` method in several interfaces implemented by a class do *not* result in multiple inheritance of implementation, as no implementation is inherited for such methods. The `abstract` method `doIt()` has only one implementation in the `Worker` class. Multiple inheritance of type is also *not* a problem, as the class is a subtype of all interfaces it implements.

If the `doIt()` methods in the two previous interfaces at (1) and (2) had the same signatures but different return types, the `Worker` class would not be able to implement both interfaces. This is illustrated by the code below. The `doIt()` methods at (1) and (2) have the same signature, but different return types. The `ChallengedWorker` class provides two implementations of the `doIt()` method at (5) and (6), which results in compile-time errors because a class cannot have two methods with the same signature but different return types. Removing either implementation from the `ChallengedWorker` class will be flagged as a compile-time error because the `ChallengedWorker` class will not be implementing both interfaces. There is no way the `ChallengedWorker` class can implement both interfaces, given the declarations shown in the code. In addition, the `abstract` class `Combined` at (3) will not compile because it will be inheriting two methods with conflicting `abstract` method declarations. In fact, the compiler complains of duplicate methods.

Click here to view code image

```
interface IA { int doIt(); }                          // (1)

interface IB { double doIt(); }                       // (2)

abstract class Combined implements IA, IB { }         // (3) Compile-time erro

public class ChallengedWorker implements IA, IB {     // (4)
  @Override
  public int doIt() { return 0; }                     // (5) Compile-time erro
  @Override
  public double doIt() {                              // (6) Compile-time erro
    System.out.println("Sorry!");
    return = 0.0;
  }
}
```

An enum type (p. 298) can also implement interfaces. The discussion above on classes implementing interfaces also applies to enum types, but with the following caveat. Since an enum type can never be `abstract`, each `abstract` method of its declared interfaces must be implemented as an instance member either in the enum type or in *all* constant-specific class bodies defined in the enum type — otherwise, the code will not compile.

Extending Interfaces

An interface can extend other interfaces, using the `extends` clause. Unlike when extending classes, an interface can extend several interfaces. The interfaces extended by an interface (directly or indirectly) are called *superinterfaces*. Conversely, the interface is a *subinterface* of its superinterfaces. Since interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes, respectively.

A subinterface inherits from its superinterfaces all members of those superinterfaces, *except* for the following:

- Any `abstract` or `default` methods that it overrides from its superinterfaces (p. 240 and p. 246)
- Any `static` methods declared in its superinterfaces (p. 251)
- Any `static` constants that it hides from its superinterfaces (p. 254)
- Any `static` member types that it hides from its superinterfaces (§9.2, p. 495)

Barring any conflicts, a subinterface inherits `abstract` and `default` method declarations that are not overridden, as well as constants and `static` member types that it does not hide in its superinterfaces. In addition, `abstract`, `static`, and `default` method declarations can be overloaded, analogous to method overloading in classes. For a detailed discussion of overriding `abstract` methods from multiple superinterfaces, see §11.12, p. 621.

Example 5.10 illustrates the relationships between classes and interfaces. In Example 5.10, the interface `ISafeStack` extends the interface `IStack` at (5). The class `SafeStack` both extends the `Stack` class and implements the `ISafeStack` interface at (6). Inheritance hierarchies for classes and interfaces defined in Example 5.10 are shown in Figure 5.4.

In UML, an interface resembles a class. One way to differentiate between them is to use an *«interface»* stereotype, as in Figure 5.4. The association between a class and any interface it implements is called a *realization* in UML. Realization is depicted in a similar manner to extending classes, but is indicated by an unbroken inheritance arrow. In Figure 5.4, there are two explicit realizations: The class `Stack` implements the `IStack` interface and the class `SafeStack` implements the `ISafeStack` interface.

It is instructive to consider how the class `SafeStack` transitively also implements the `IStack` interface in Example 5.10. This is evident from the diamond shape of the inheritance hierarchy in Figure 5.4. The class `SafeStack` inherits the implementations of the `push()` and `pop()` methods from its superclass `Stack`, which itself implements the

Figure 5.4 *Inheritance Hierarchies for Classes and Interfaces in*

Example 5.10

`IStack` interface in which these two methods are declared. The class `SafeStack` also implements the `IStack` interface via the `ISafeStack` interface. The class `SafeStack` provides its own implementation of the `isFull()` and `isEmpty()` methods declared in the `ISafeStack` interface, and has inherited implementations of the `push()` and `pop()` methods whose declarations the `ISafeStack` interface inherits from its super-interface `IStack`. Note that there is only one *implementation* of a method that is inherited by the class `SafeStack` —from its superclass `Stack`.

Thinking in terms of types, every reference type in Java is a subtype of the `java.lang.Object` class. In turn, any interface type is also a subtype of the `Object` class, but it does *not* inherit any implementation from the `Object` class. As mentioned earlier, an interface that has *no direct superinterfaces* implicitly declares a `public abstract` method for *each* `public` instance method in the `Object` class. These `abstract` method declarations are inherited by all subinterfaces of such an interface. Note that this does not mean any *implementation* is inherited by the subinterfaces. The implicit `public abstract` method declarations in an interface allow `public` instance methods in the `Object` class to be invoked on objects referenced by an interface reference. All classes implement these methods, whether they are inherited or overridden from the `Object` class. Any interface can also provide *explicit* `public abstract` method declarations for *non-final* `public` instance methods in the `Object` class.

Click here to view code image

```
interface IStack {                                                 // (1)
  void   push(Object item);
  Object pop();
  @Override boolean equals(Object other);         // public method in Object cl
  @Override String toString();                    // public method in Object cl
//@Override Class getClass(); // Compile-time error! final method in Object cl
}
```

Interface References

Although interfaces cannot be instantiated, references of an interface type can be declared. The reference value of an object can be assigned to references of the object’s supertypes. In Example 5.10, an object of the class `SafeStack` is created in the `main()` method of the class `StackUser` at (9). The reference value of the object is assigned to references of all the object’s supertypes, which are used to manipulate the object. The references are aliases to the same `SafeStack` object, but they can only be used to manipulate this object as an object of the reference type. For example, calling the method `isFull()` on this object using the `stackRef` reference will be flagged as a compile-time error, as the class `Stack` does not provide such a method.

Polymorphic behavior of supertype references is discussed later in this chapter (p. 278).

Default Methods in Interfaces

Only interfaces can define `default` methods. A `default` method is an *instance method* declared with the keyword `default` and whose implementation is provided by the interface. However, a `default` method in a top-level interface always has `public` access, whether the keyword `public` is specified or not.

Click here to view code image

```
default return_type method_name (formal_parameter_list) throws_clause {
    implementaion_of_method_body
}
```

A class implementing an interface can optionally decide to override any `default` method in the interface, as can a subinterface of the interface. If a `default` method is not overridden to provide a new implementation, the `default` implementation provided by the interface is inherited by the class or the subinterface.

No other non-access modifiers, such as `abstract`, `final`, or `static`, are allowed in a `default` method declaration. A `default` method is not `abstract` because it provides an implementation; is not `final` because it can be overridden; and is not `static` because it can be invoked only on instances of a class that implements the interface in which the `default` method is declared.

Note that a `default` method can only be implemented in terms of the values of its local variables (including those passed as parameters) and calls to other methods accessible in the interface, but without reference to any persistent state, as there is none associated with an interface.

The keyword `default` in the context of a `default` method should not be confused with default or package accessibility of a method in a class, which is implied in the absence of any access modifier.

Example 5.11 illustrates the use of `default` methods. The `default` method `printSlogan()` at (1) in the interface `ISlogan` is overridden at (2) in the class `JavaGuru`, and is inherited by the class `JavaGeek` at (3). The output from the program shows that this is the case.

Example 5.11 *Default Methods in Interfaces*

Click here to view code image

```
// File: JavaParty.java
interface ISlogan {
  default void printSlogan() {                       // (1)
    System.out.println("Happiness is getting certified!");
  }
}
//____________________________________________________________________________
class JavaGuru implements ISlogan {
  @Override
  public void printSlogan() {                        // (2) overrides (1)
    System.out.println("Happiness is catching all the exceptions!");
  }
}
//____________________________________________________________________________
class JavaGeek implements ISlogan { }                // (3) inherits (1)
//____________________________________________________________________________
public class JavaParty {
  public static void main(String[] args) {
    JavaGuru guru = new JavaGuru();
    guru.printSlogan();                              // (4)
    JavaGeek geek = new JavaGeek();
    geek.printSlogan();                              // (5)
  }
}
```

Output from the program:

Click here to view code image

```
Happiness is catching all the exceptions!
Happiness is getting certified!
```

Overriding Default Methods

Overriding a `default` method from an interface does not necessarily imply that a new implementation is being provided. The `default` method can also be overridden by providing an `abstract` method declaration, as illustrated by the code below. The `default` method `printSlogan()` at (1) in the interface `ISlogan` is overridden by an `abstract` method declaration at (2) and (3) in the interface `INewSlogan` and the `abstract` class `JavaMaster`, respectively. This strategy effectively forces the subtypes of the interface `INewSlogan` and of the `abstract` class `JavaMaster` to provide a new concrete implementation for the method, as one would expect for an `abstract` method.

Click here to view code image

```
interface ISlogan {
  default void printSlogan() {          // (1) Default method.
    System.out.println("Happiness is getting certified!");
  }
}

interface INewSlogan extends ISlogan {
  @Override
  abstract void printSlogan();          // (2) overrides (1) with abstract meth
}

abstract class JavaMaster implements ISlogan {
  @Override
  public abstract void printSlogan();  // (3) overrides (1) with abstract meth
}
```

Conflict Resolution for Default Methods

Conflicts with multiple inheritance of implementation can arise when `default` methods are inherited from unrelated interfaces.

Example 5.12 illustrates the case where two interfaces define a `default` method with the same signature. The `default` method `printSlogan()` is declared at (1) and (2) in the interfaces `ICheapSlogan` and `IFunnySlogan`, respectively. The two method declarations have the same signature. The interface `IAvailableSlogan` at (3) tries to extend the two interfaces `ICheapSlogan` and `IFunnySlogan`. If this were allowed, the interface `IAvailableSlogan` would inherit two implementations of a method with the same signature, which of course is not allowed—so the compiler flags it as an error. By the same token, the compiler flags an error at (4), indicating that the `abstract` class `Wholesaler` cannot inherit two `default` methods with the same signature.

A way out of this dilemma is to override the conflicting `default` methods. The `abstract` class `RetailSeller` that implements the interfaces `ICheapSlogan` and `IFunnySlogan` overrides the conflicting methods by providing an `abstract` method declaration of the `default` method `printSlogan()` at (5). Similarly, the class `NetSeller` that implements the interfaces `ICheapSlogan` and `IFunnySlogan` overrides the conflicting methods by providing an implementation of the `default` method `printSlogan()` at (6).

The upshot of this solution is that clients of the classes `RetailSeller` and `NetSeller` now have to deal with the new declarations of the `printSlogan()` method provided by these classes. One such client is the class `MultipleInheritance` at (10), which calls the method `printSlogan()` on an instance of class `NetSeller` at (11). Not surprisingly, the program output shows that the method in the `NetSeller` class was executed.

What if the class `NetSeller` wanted to invoke the `default` method `printSlogan()` in the interfaces it implements? The overridden `default` method can be called by the overriding subtype (in this case, `NetSeller`) using the keyword `super` in conjunction with the fully qualified name of the interface and the name of the method, as shown at (8) and (9). This syntax works for calling overridden `default` methods in the *direct* superinterface, but not at any higher level in the inheritance hierarchy. The class `NetSeller` can call only `default` methods in its direct superinterfaces `ICheapSlogan` and `IFunnySlogan`. It would not be possible for the class `NetSeller` to call any `default` methods inherited by these superinterfaces, even if they had any.

Example 5.12 *Inheriting Default Method Implementations from Superinterfaces*

Click here to view code image

```
// File: MultipleInheritance.java
interface ICheapSlogan {
  default void printSlogan() {          // (1)
    System.out.println("Override, don't overload.");
  }
}
//_____________________________________________________________________________
interface IFunnySlogan {
  default void printSlogan() {          // (2)
    System.out.println("Catch exceptions, not bugs.");
  }
}
//_____________________________________________________________________________
interface IAvailableSlogan              // (3) Compile-time error.
          extends ICheapSlogan, IFunnySlogan { }
//_____________________________________________________________________________
abstract class Wholesaler               // (4) Compile-time error.
               implements ICheapSlogan, IFunnySlogan { }
//_____________________________________________________________________________
abstract class RetailSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Abstract method.
  public abstract void printSlogan();   // (5) overrides (1) and (2).
}
//_____________________________________________________________________________
class NetSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Concrete method.
  public void printSlogan() {           // (6) overrides (1) and (2).
    System.out.println("Think outside of the class.");
  }

  public void invokeDirect() {          // (7)
```

```
    ICheapSlogan.super.printSlogan();   // (8) calls ICheapSlogan.printSlogan(
    IFunnySlogan.super.printSlogan();   // (9) calls IFunnySlogan.printSlogan(
  }
}
//____________________________________________________________________________
public class MultipleInheritance {      // (10)
  public static void main(String[] args) {
    NetSeller seller = new NetSeller();
    seller.printSlogan();               // (11)
    seller.invokeDirect();
  }
}
```

Output from the program:

Click here to view code image

```
Think outside of the class.
Override, don't overload.
Catch exceptions, not bugs.
```

Example 5.13 illustrates the case where a concrete method at (1) in the class `Slogan` and a `default` method at (2) in the interface `ISlogan` have the same signature. The subclass `MySlogan` at (3) extends the superclass `Slogan` and implements the super-interface `ISlogan`. The class `MySlogan` compiles even though it looks like two implementations of the `printSlogan()` method are being inherited. In this special case there is no multiple inheritance of implementation, as only the implementation from the superclass `Slogan` is inherited. The implementation of the `default` method at (2) is ignored. This is borne out by the program output when the method `print-Slogan()` is called at (5) on an object of class `MySlogan`.

Interface Evolution

Augmenting an existing interface with an `abstract` method will break all classes that implement this interface, as they will no longer implement the old interface. These classes will need to be modified and recompiled in order to work with the augmented interface.

Augmenting an existing interface with a `default` method does not pose this problem. Classes that implement the old interface will work with the augmented interface *without modifying or recompiling* them. Thus interfaces can evolve without affecting classes that worked with the old interface. This is also true for `static` methods (p. 251) added to existing interfaces.

Example 5.13 *Inheriting Method Implementations from Supertypes*

Click here to view code image

```
// File: MultipleInheritance2.java
class Slogan {
  public void printSlogan() {                         // (1) Concrete method
    System.out.println("Superclass wins!");
  }
}
//______________________________________________________________________________
interface ISlogan {
  default void printSlogan() {                        // (2) Default method
    System.out.println("Superinterface wins!");
  }
}
//______________________________________________________________________________
class MySlogan extends Slogan implements ISlogan { } // (3)
//______________________________________________________________________________
public class MultipleInheritance2 {                   // (4)
  public static void main(String[] args) {
    MySlogan slogan = new MySlogan();
    slogan.printSlogan();                             // (5)
  }
}
```

Output from the program:

```
Superclass wins!
```

Static Methods in Interfaces

A common practice in designing APIs has been to provide an interface that classes can implement and a separate *utility class* providing `static` methods for common operations on objects of these classes. Typical examples are the `java.util.Collection` interface and the `java.util.Collections` utility class (Chapter 15, p. 781). Another example is the `Path` interface and the `Paths` utility class in the `java.nio.file` package (Chapter 21, p. 1285). However, now an interface can also declare `static` methods, and there is no need for a separate utility class.

Static method declarations in a top-level interface are declared analogous to `static` method declarations in a class. However, a `static` method in a top-level interface always has `public` access, whether the keyword `public` is specified or not. As with `static` methods in a class, the keyword `static` is mandatory; otherwise, the code will not compile. Without the keyword `static`, the method declaration is identical to that of an instance method, but such instance methods cannot be declared in an interface and the compiler will flag an error.

Click here to view code image

```
static return_type method_name  (formal_parameter_list)  throws_clause  {
      implementaion_of_method_body
}
```

Static methods in an interface differ from those in a class in one important respect: Static methods in an interface *cannot* be inherited, unlike `static` methods in classes. This essentially means that such methods cannot be invoked directly by calling the method in subinterfaces or in classes that extend or implement interfaces containing such methods, respectively. A `static` method can be invoked only by using its qualified name—that is, the name of the interface in which it is declared, together with its simple name, using the dot notation (`.`).

Example 5.14 illustrates the use of `static` methods in interfaces. The `static` method `getNumOfCylinders()` at (1) is declared in the `IMaxEngineSize` interface. There are two implementations of the method `getEngineSize()`, at (2) and (3), in the interface `IMaxEngineSize` and its subinterface `INewEngineSize`, respectively. The class `CarRace` implements the subinterface `INewEngineSize`.

It is not possible to invoke the `static` method `getNumOfCylinders()` directly, as shown at (4). It is also not possible to invoke directly the `static` method `getEngineSize()` from either interface, as shown at (6). The respective implementations of the `static` methods can be invoked only by using their qualified names, as shown at (5), (7), and (8). It does not matter that a `static` method is redeclared in a subinterface; the `static` method is not inherited. Each `static` method declaration in Example 5.14 is a new method.

Example 5.14 *Static Methods in Interfaces*

Click here to view code image

```
// File: CarRace.java
import static java.lang.System.out;
interface IMaxEngineSize {
  static int getNumOfCylinders() { return 6; }         // (1) Static method
  static double getEngineSize() { return 1.6; }        // (2) Static method
}
//______________________________________________________________________________
interface INewEngineSize extends IMaxEngineSize {
  static double getEngineSize() { return 2.4; }        // (3) Static method
}
//______________________________________________________________________________
public class CarRace implements INewEngineSize {
  public static void main(String[] args) {
```

```
//  out.println("No. of cylinders: " +
//               getNumOfCylinders());                  // (4) Compile-time erro
    out.println("No. of cylinders: " +
        IMaxEngineSize.getNumOfCylinders());           // (5)
//  out.println("Engine size: " + getEngineSize());    // (6) Compile-time erro
    out.println("Max engine size: " + IMaxEngineSize.getEngineSize()); // (7)
    out.println("New engine size: " + INewEngineSize.getEngineSize()); // (8)
  }
}
```

Output from the program:

```
No. of cylinders: 6
Max engine size: 1.6
New engine size: 2.4
```

Private Methods in Interfaces

Private methods in interfaces are no different from `private` methods in classes. As such, they can only be accessed inside the interface, acting as *helper* or *auxiliary methods* for non-`abstract` methods declared in the interface. They allow code to be shared between the non-`abstract` methods in the interface, thus promoting code reuse and avoiding code duplicity.

Private methods in an interface are governed by the following rules:

- Not surprisingly, a `private` method must be declared with the `private` modifier in its header.
- A `private` method can only be accessed inside the interface itself.
- A `private` method cannot be declared with the `abstract` or the `default` modifier, but can be declared either as a `private` instance method or a `private static` method.
- A `private static` method can be invoked in any non-`abstract` method in the interface.
- A `private` instance method can only be invoked in `default` and `private` instance methods of the interface, and not in any `static` methods of the interface.

The above rules stem from accessibility of a `private` member in a type declaration in which the code is accessible in static and non-static contexts.

Example 5.15 illustrates declaring `private` methods in an interface. The `default` method `getSalePrice()` at (2) and the `static` method `startSale()` at (5) call the `private` instance method `wrapUp()` at

(4) and the `private static` method `showSaleItems()` at (7), respectively.

Example 5.15 *Private Methods in Interfaces*

Click here to view code image

```
/** Interface with private methods. */
public interface IShopper {

  // Abstract method:
  double getItemPrice();                               // (1)

  // Default method:
  default double getSalePrice(double price) {          // (2)
    var salePrice = (80.0/100.0)*price;
    System.out.println("Default method: " + "Sale price is " + salePrice);
    wrapUp();                    // (3) Calls the private instance method at (
    return salePrice;
  }

  // Private instance method:
  private void wrapUp() {                               // 4)
    System.out.println("Private method: " + "Wrapping up!");
  }

  // Static method:
  static void startSale() {                             // (5)
    System.out.println("Static method: " + "Amazing savings!");
    showSaleItems();             // (6) Calls the private static method at (7)
  }

  // Private static method:
  private static void showSaleItems() {                 // (7)
    System.out.println("Private static method: " + "Sorry. No items on sale!")
  }
}
```

```
/** Class Shopper implements IShopper interface */
public class Shopper implements IShopper {

  @Override
  public double getItemPrice() {
    return 100.00;
  }

  public static void main(String[] args) {
    Shopper customer = new Shopper();
    var price = customer.getItemPrice();
    System.out.println("Item price: " + price);
    var salePrice = customer.getSalePrice(price); // Calls default method at (
    System.out.println();
    IShopper.startSale();                         // Calls static method at (5
  }
}
```

Output from the program:

Click here to view code image

```
Item price: 100.0
Default method: Sale price is 80.0
Private method: Wrapping up!
```

```
Static method: Amazing savings!
Private static method: Sorry. No items on sale!
```

Constants in Interfaces

A field declaration in an interface defines a *named constant*. Naming conventions recommend using uppercase letters, with multiple words in the name being separated by underscores. Such constants are considered to be `public`, `static`, and `final`. These modifiers are usually omitted from the declaration, but can be specified in any order. Such a constant must be initialized with an initializer expression.

An interface constant can be accessed by any client (a class or interface) using its qualified name, regardless of whether the client extends or implements its interface. However, if the client is a class that implements this interface or is an interface that extends this interface, then the client can also access such constants directly by their simple names. Such a client inherits the interface constants. Typical usage of constants in interfaces is illustrated in Example 5.16, showing access both by the constant's simple name and its qualified name in the print statements at (1) and (2), respectively.

Example 5.16 *Constants in Interfaces*

Click here to view code image

```
// File: Client.java
interface Constants {
  double PI_APPROXIMATION = 3.14;
  String AREA_UNITS       = "sq.cm.";
  String LENGTH_UNITS     = "cm.";
}
//____________________________________________________________________________
public class Client implements Constants {
  public static void main(String[] args) {
    double radius = 1.5;

    // (1) Using simple name:
    System.out.printf("Area of circle is %.2f %s%n",
                 PI_APPROXIMATION * radius*radius, AREA_UNITS);

    // (2) Using qualified name:
    System.out.printf("Circumference of circle is %.2f %s%n",
               2.0 * Constants.PI_APPROXIMATION * radius, Constants.LENGTH_UNITS
  }
}
```

Output from the program:

Click here to view code image

```
Area of circle is 7.06 sq.cm.
Circumference of circle is 9.42 cm.
```

Extending an interface that has constants is analogous to extending a class that has `static` variables. This is illustrated in Figure 5.5 and Example 5.17. Note the diamond shape of the inheritance hierarchy, indicating the presence of multiple inheritance paths through which constants can be inherited. The constants `IDLE` and `BUSY` at (1) and (2) in the interface `IBaseStates` are inherited by the subinterface `IAllStates` via both the interface `IExtStatesA` and the interface `IExtStatesB`. In such cases, the constant is considered to be inherited only once and can be accessed by its simple name, as shown at (12) in Example 5.17.

Figure 5.5 *Inheritance Relationships for Interface Constants*

Constants can be *hidden* by the subinterfaces. The declaration of the constant `BLOCKED` at (6) in the interface `IAllStates` hides the declaration of the constant at (2) in the interface `IBaseStates`. The new declaration can be accessed by its simple name in a class implementing the interface `IAllStates`, as shown at (10) in Example 5.17. The hidden constant declaration can always be accessed by using its qualified name as shown at (11) in Example 5.17.

In the case of multiple inheritance of interface constants, any name conflicts can be resolved by using the qualified name to access the constants. This is illustrated by the constant `DISMANTLED`, which is declared in both the `IExtStatesA` and `IExtStatesB` interfaces. Both declarations are inherited by the subinterface `IAllStates`. Such declarations are said to be *ambiguous*. The compiler will report an error only if such constants are accessed by their simple names, as shown at (7) and (8) for the constant `DISMANTLED`. Only the qualified name can be used to disambiguate such constants and resolve the conflict, as shown at (7a) and (8a) for the constant `DISMANTLED`.

When defining a *set of related constants*, the recommended practice is to use an enum type, rather than named constants in an interface.

Example 5.17 *Inheriting Constants in Interfaces*

Click here to view code image

```
// File: Factory.java
import static java.lang.System.out;

interface IBaseStates {
  String IDLE = "idle";                                // (1)
  String BUSY = "busy";                                // (2)
  String BLOCKED = "blocked";                          // (3)
}
//_____________________________________________________________________________
interface IExtStatesA extends IBaseStates {
  String DISMANTLED = "dismantled";                    // (4)
}
//_____________________________________________________________________________
interface IExtStatesB extends IBaseStates {
  String DISMANTLED = "kaput";                         // (5)
}
//_____________________________________________________________________________
interface IAllStates extends IExtStatesB, IExtStatesA {
  String BLOCKED = "out of order";                     // (6) hides (3)
//String OBSOLETE = BLOCKED + ", " +
//                  DISMANTLED + " and scrapped.";     // (7) Ambiguous
  String OBSOLETE = BLOCKED + ", " +
         IExtStatesB.DISMANTLED + " and scrapped";     // (7a)
}
//_____________________________________________________________________________
public class Factory implements IAllStates {
  public static void main(String[] args) {
//  out.println("Machine A is " + DISMANTLED);             // (8) Ambiguous.
    out.println("Machine A is " + IExtStatesB.DISMANTLED); // (8a)
    out.println("Machine B is " + OBSOLETE);           // (9) IAllStates.OBSOL
    out.println("Machine C is " + BLOCKED);            // (10) IAllStates.BLOC
    out.println("Machine D is " + IBaseStates.BLOCKED);// (11)
    out.println("Machine E is " + BUSY);               // (12) Simple name
  }
}
```

Output from the program:

Click here to view code image

```
Machine A is kaput
Machine B is out of order, kaput and scrapped
Machine C is out of order
Machine D is blocked
Machine E is busy
```

Review Questions

5.18 Which modifiers are not allowed for methods in an interface ? Select the two correct answers.

a. `public`
b. `protected`
c. `private`
d. `default`
e. `abstract`

f. `static`

g. `final`

5.19 Which method call can be inserted at both (1) and (2) so that the following code will still compile?

Click here to view code image

```
interface INewSlogan {
  String SLOGAN = "Trouble shared is trouble halved!";
  static void printSlogan() { System.out.println(SLOGAN); }
}
//_____________________________________________________________________
public class Firm implements INewSlogan {
  public static void main(String[] args) {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (1) INSERT STATEMENT EXPRESSION HERE
  }

  void testSlogan() {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (2) INSERT STATEMENT EXPRESSION HERE
  }
}
```

Select the one correct answer.

a. `printSlogan();`

b. `co.printSlogan();`

c. `sl.printSlogan();`

d. `Firm.printSlogan();`

e. `INewSlogan.printSlogan();`

5.20 What will be the result of compiling and running the following program?

Click here to view code image

```
interface IJogger {
  default boolean justDoIt(String msg) { return false; }  // (1)
  static  boolean justDoIt(int i)      { return true; }   // (2)
}

class Athlete implements IJogger {
  public boolean justDoIt(String msg)  { return true; }   // (3)
  public boolean justDoIt(int i)       { return false; }  // (4)
}

public class RaceA {
  public static void main(String[] args) {
    Athlete athlete = new Athlete();
    IJogger jogger = athlete;
    System.out.print(jogger.justDoIt("Run"));             // (5)
    System.out.println("|" + athlete.justDoIt(10));       // (6)
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. `true|true`

c. `true|false`

d. `false|true`

e. `false|false`

5.21 Which statement is true about the following code?

Click here to view code image

```
abstract class MyClass implements Interface1, Interface2 {
  public void f() { }
  public void g() { }
}

interface Interface1 {
  int VAL_A = 1;
  int VAL_B = 2;

  void f();
  void g();
}

interface Interface2 {
  int VAL_B = 3;
  int VAL_C = 4;

  void g();
  void h();
}
```

Select the one correct answer.

a. `MyClass` implements only `Interface1`; the implementation for `void h()` from `Interface2` is missing.

b. The declarations of `void g()` in the two interfaces are in conflict, so the code will fail to compile.

c. The declarations of `int VAL_B` in the two interfaces are in conflict, so the code will fail to compile.

d. Nothing is wrong with the code; it will compile without errors.

5.7 Arrays and Subtyping

Table 5.4 summarizes the types found in Java. Only primitive data and reference values can be stored in variables. Only class and array types can be explicitly instantiated to create objects.

Table 5.4 *Types and Values*

Types	Values
Primitive data types	Primitive data values
Class, interface, enum, and array types (*reference types*)	Reference values

Arrays and Subtype Covariance

Arrays are objects in Java. Array types (`boolean[]`, `Object[]`, `Stack[]`) implicitly augment the inheritance hierarchy. The inheritance hierarchy depicted in Figure 5.4, for example, can be augmented by the corresponding array types to produce the *type hierarchy* shown in Figure 5.6. An array type is shown as a "class" with the `[]` notation appended to the name of the element type. The class `SafeStack` is a subclass of the class `Stack`. The corresponding array types, `SafeStack[]` and `Stack[]`, are shown as the subtype and the supertype, respectively, in the type hierarchy. Figure 5.6 also shows array types corresponding to some of the primitive data types.

Figure 5.6 *Reference Type Hierarchy: Arrays and Subtype Covariance*

From the type hierarchy in Figure 5.6, the following facts are apparent:

- *All* reference types are subtypes of the `Object` type. This applies to classes, interfaces, enums, and array types, as these are all reference types.
- All arrays of reference types are also subtypes of the array type `Object[]`, but arrays of primitive data types are not. Note that the array type `Object[]` is also a subtype of the `Object` type.
- If a non-generic reference type is a subtype of another non-generic reference type, the corresponding array types also have an analogous subtype–supertype relationship. This is called the *subtype covariance relationship*.
- There is no subtype–supertype relationship between a type and its corresponding array type.

We can create an array of an interface type, but we cannot instantiate an interface (as is the case with `abstract` classes). In the following declaration statement, the reference `arrayOfISafeStack` has type `ISafeStack[]` (i.e., an array of the interface type `ISafeStack`):

Click here to view code image

```
ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
```

The array creation expression creates an array whose element type is `ISafeStack`. The array object can accommodate five references of the type `ISafeStack`. The declaration statement does not initialize these references to refer to any objects; instead, they are initialized to the default value `null`.

Array Store Check

An array reference exhibits polymorphic behavior like any other reference, subject to its location in the type hierarchy (p. 278). However, a runtime check is necessary when objects are inserted in an array, as illustrated below.

The following assignment is valid, as a supertype reference (`Stack[]`) can refer to objects of its subtype (`SafeStack[]`):

Click here to view code image

```
Stack[] arrayOfStack = new SafeStack[2];      // (1)
```

Since `Stack` is a supertype of `SafeStack`, the following assignment is also valid:

Click here to view code image

```
arrayOfStack[0] = new SafeStack(10);          // (2)
```

The assignment at (2) assigns the reference value of a new `SafeStack` object to the reference at index `0` in the `SafeStack[]` object created at (1).

Since the type of `arrayOfStack[i]`, (`0` £ `i` `<` `2`), is `Stack`, it should be possible to make the following assignment as well:

Click here to view code image

```
arrayOfStack[1] = new Stack(20);              // (3) ArrayStoreException
```

At compile time there are no problems, as the compiler cannot deduce that the array variable `arrayOfStack` will actually denote a `SafeStack[]` object at runtime. However, the assignment at (3) results in an `ArrayStoreException` being thrown at runtime because an array of `SafeStack` objects cannot possibly contain objects of its supertype `Stack`.

The array store check at runtime ensures that an object being stored in the array is assignment compatible (p. 264) with the element type of the array. To make the array store check feasible at runtime, the array retains information about its declared element type at runtime.

5.8 Reference Values and Conversions

A review of conversions (§2.3, p. 43) is recommended before proceeding with this section.

Reference values, like primitive values, can be assigned, cast, and passed as arguments. Conversions can occur in the following contexts:

- Assignment
- Method invocation
- Casting

The rule of thumb for the primitive data types is that widening conversions are permitted, but narrowing conversions require an explicit cast. The rule of thumb for reference values is that widening conversions up the type hierarchy are permitted, but narrowing conversions down the hierarchy require an explicit cast. In other words, conversions that are from a subtype to its supertypes are allowed, but other conversions require an explicit cast or are otherwise illegal. There is no notion of promotion for reference values.

5.9 Reference Value Assignment Conversions

In the context of assignments, the following conversions are permitted (Table 2.17, p. 47):

- Widening primitive and reference conversions (`long` ← `int`, `Object` ← `String`)
- Boxing conversion of primitive values, followed by optional widening reference conversion (`Integer` ← `int`, `Number` ← `Integer` ← `int`)
- Unboxing conversion of a primitive value wrapper object, followed by optional widening primitive conversion (`long` ← `int` ← `Integer`)

In addition, for assignment conversions only, the following conversion is also possible:

- Narrowing conversion for constant expressions of non-`long` integer types, with optional boxing (`Byte` ← `byte` ← `int`)

Note that these rules imply that a widening conversion *cannot* be followed by any boxing conversion, but the converse is permitted.

Widening reference conversions typically occur during assignment *up* the type hierarchy, with implicit conversion of the source reference value to that of the destination reference type:

Click here to view code image

```
Object obj = "Up the tree";    // Widening reference conversion: Object <-- St
String str1 = obj;       // Not OK. Narrowing reference conversion requires a
String str2 = Integer.valueOf(10); // Illegal. No relation between
                                   //           String and Integer.
```

The source value can be a primitive value, in which case the value is boxed in a wrapper object corresponding to the primitive type. If the destination reference type is a supertype of the wrapper type, a widening reference conversion can occur:

Click here to view code image

```
Integer iRef = 10;  // Only boxing
Number num = 10L;   // Boxing, followed by widening: Number <--- Long <--- lon
Object obj = 100;   // Boxing, followed by widening: Object <--- Integer <---
```

More examples of boxing during assignment can be found in §2.3, p. 45.

Example 5.18 *Assigning and Passing Reference Values*

Click here to view code image

```
// See Example 5.10, p. 241, for type declarations.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

Click here to view code image

```
public class ReferenceConversion {

  public static void main(String[] args) {
    // Reference declarations:
    Object     objRef;
    Stack      stackRef;
    SafeStack  safeStackRef;
    IStack     iStackRef;
    ISafeStack iSafeStackRef;

    // SourceType is a class type:
    safeStackRef  = new SafeStack(10);
    objRef        = safeStackRef;    // (1) Always possible
    stackRef      = safeStackRef;    // (2) Subclass to superclass assignment
    iStackRef     = stackRef;        // (3) Stack implements IStack
    iSafeStackRef = safeStackRef;    // (4) SafeStack implements ISafeStack

    // SourceType is an interface type:
    objRef     = iStackRef;           // (5) Always possible
    iStackRef = iSafeStackRef;       // (6) Sub- to superinterface assignment

    // SourceType is an array type:
    Object[]     objArray          = new Object[3];
    Stack[]      arrayOfStack      = new Stack[3];
    SafeStack[]  arrayOfSafeStack  = new SafeStack[5];
    ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
    int[]        intArray          = new int[10];

    // Reference value assignments:
    objRef      = objArray;           // (7) Always possible
    objRef      = arrayOfStack;       // (8) Always possible
    objArray    = arrayOfStack;       // (9) Always possible
    objArray    = arrayOfISafeStack;  // (10) Always possible
    objRef      = intArray;           // (11) Always possible
    //  objArray   = intArray;        // (12) Compile-time error:
                                      //      int[] not subtype of Object[]
    arrayOfStack = arrayOfSafeStack; // (13) Subclass array to superclass arra
    arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implements
                                          //      ISafeStack

    // Method invocation conversions:
    System.out.println("First call:");
    sendParams(stackRef, safeStackRef, iStackRef,
               arrayOfSafeStack, arrayOfISafeStack);                  // (15)
    //  Call Signature: sendParams(Stack, SafeStack, IStack,
    //                             SafeStack[], ISafeStack[]);

    System.out.println("Second call:");
    sendParams(arrayOfISafeStack, stackRef, iSafeStackRef,
               arrayOfStack, arrayOfSafeStack);                       // (16)
    //  Call Signature: sendParams(ISafeStack[], Stack, ISafeStack,
    //                             Stack[], SafeStack[]);
  }

  public static void sendParams(Object objRefParam, Stack stackRefParam,
      IStack iStackRefParam, Stack[] arrayOfStackParam,
      IStack[] arrayOfIStackParam) {                                    // (17
    //  Signature: sendParams(Object, Stack, IStack, Stack[], IStack[])
    //  Print class name of object denoted by the reference at runtime.
    System.out.println(objRefParam.getClass());
    System.out.println(stackRefParam.getClass());
    System.out.println(iStackRefParam.getClass());
    System.out.println(arrayOfStackParam.getClass());
    System.out.println(arrayOfIStackParam.getClass());
  }
}
```

Output from the program:

Click here to view code image

```
First call:
class SafeStack
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
Second call:
class [LSafeStack;
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
```

The rules for reference value assignment are stated in this section, based on the following code:

Click here to view code image

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

If an assignment is legal, the reference value of `srcRef` is said to be *assignable* (or *assignment compatible*) to the reference of `DestinationType`. The rules are illustrated by concrete cases from Example 5.18. Note that the code in Example 5.18 uses reference types from Example 5.10, p. 241.

- If the `SourceType` is a *class type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:
    - `DestinationType` is a superclass of the subclass `SourceType`.
    - `DestinationType` is an interface type that is implemented by the class `SourceType`.

  Click here to view code image

  ```
  objRef         = safeStackRef;     // (1) Always possible
  stackRef       = safeStackRef;     // (2) Subclass to superclass
  iStackRef      = stackRef;         // (3) Stack implements IStac
  iSafeStackRef  = safeStackRef;     // (4) SafeStack implements I
  ```

- If the `SourceType` is an *interface type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:
    - `DestinationType` is the `Object` class.
    - `DestinationType` is a superinterface of the subinterface `SourceType`.

  Click here to view code image

  ```
  objRef    = iStackRef;      // (5) Always possible
  iStackRef = iSafeStackRef;  // (6) Subinterface to superinterfa
  ```

- If the `SourceType` is an *array type*, the reference value in `srcRef` may be assigned to the `destRef` reference, provided the `DestinationType` is one of the following:

- `DestinationType` is the `Object` class.
- `DestinationType` is an array type, where the element type of the `SourceType` is assignable to the element type of the `DestinationType`.

Click here to view code image

```
objRef      = objArray;           // (7) Always possible
objRef      = arrayOfStack;       // (8) Always possible
objArray    = arrayOfStack;       // (9) Always possible
objArray    = arrayOfISafeStack;  // (10) Always possible
objRef      = intArray;           // (11) Always possible
// objArray    = intArray;        // (12) Compile-time error:
                                  //      int[] not subtype of
arrayOfStack = arrayOfSafeStack;  // (13) Subclass array to sup
arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implem
                                      //      ISafeStack
```

The rules for assignment are enforced at compile time, guaranteeing that no type conversion error will occur during assignment at runtime. Such conversions are *type-safe*. The reason the rules can be enforced at compile time is that they concern the *declared type* of the reference (which is always known at compile time) rather than the actual type of the object being referenced (which is known at runtime).

5.10 Method Invocation Conversions Involving References

The conversions for reference value assignment are also applicable to *method invocation conversions*, except for the narrowing conversion for constant expressions of non-`long` integer type (Table 2.17, p. 47). This is reasonable, as parameters in Java are passed by value (§3.10, p. 127), requiring that values of the actual parameters must be assignable to formal parameters of the compatible types.

In Example 5.18, the method `sendParams()` at (17) has the following signature, showing the types of the formal parameters:

Click here to view code image

```
sendParams(Object, Stack, IStack, Stack[], IStack[])
```

The method call at (15) has the following signature, showing the types of the actual parameters:

Click here to view code image

```
sendParams(Stack, SafeStack, IStack, SafeStack[], ISafeStack[]);   // Compile
```

At runtime, the actual type of the objects whose reference values are passed at (15) is the following:

Click here to view code image

```
sendParams(SafeStack, SafeStack, SafeStack, SafeStack[], SafeStack[]);  // Run
```

Note that the assignment of the values of the actual parameters to the corresponding formal parameters is legal, according to the rules for assignment discussed earlier. The method call at (16) provides another example of the parameter-passing conversion. It has the following signature:

Click here to view code image

```
sendParams(ISafeStack[], Stack, ISafeStack, Stack[], SafeStack[]); // Compile
```

At runtime, the actual type of the objects whose reference values are passed at (16) is the following:

Click here to view code image

```
sendParams(SafeStack[], SafeStack, SafeStack, SafeStack[], SafeStack[]);// Run
```

Analogous to assignment, the rules for parameter-passing conversions are based on the reference type of the parameters and are enforced at compile time. It is instructive to compare the type of a formal parameter with the type of its corresponding actual parameter at compile time, and the type of the object whose reference value is actually passed at runtime. The output in Example 5.18 shows the class of the actual objects referenced by the formal parameters at runtime, which in this case turns out to be either `SafeStack` or `SafeStack[]`. The characters `[L` in the output indicate a one-dimensional array of a class or interface type (see the `Class.getName()` method in the Java SE Platform API documentation).

Overloaded Method Resolution

In this subsection, we take a look at some aspects regarding *overloaded method resolution*—that is, how the compiler determines which overloaded method will be invoked by a given method call at runtime.

Resolution of overloaded methods selects the *most specific* method for execution. One method is considered more specific than another method if all actual parameters that can be accepted by the one method can be accepted by the other method. If more than one such method is present, the call is described as *ambiguous*. The following overloaded methods illustrate this situation:

Click here to view code image

```
private static void flipFlop(String str, int i, Integer iRef) { // (1)
    out.println(str + " ==> (String, int, Integer)");
}
private static void flipFlop(String str, int i, int j) {        // (2)
    out.println(str + " ==> (String, int, int)");
}
```

Their method signatures are as follows:

Click here to view code image

```
flipFlop(String, int, Integer)                             // See (1)
flipFlop(String, int, int)                                 // See (2)
```

The following method call is ambiguous:

Click here to view code image

```
flipFlop("(String, Integer, int)",
         Integer.valueOf(4), 2020);                        // (3) Ambiguous ca
```

It has the call signature:

Click here to view code image

```
flipFlop(String, Integer, int)                             // See (3)
```

The method at (1) can be called with the second argument unboxed and the third argument boxed, as can the method at (2) with only the second argument unboxed. In other words, for the call at (3), none of the methods is more specific than the other.

Example 5.19 illustrates a simple case of how method resolution is done to choose the most specific overloaded method. The method `testIfOn()` is overloaded at (1) and (2) in the class `Overload`. The call `client.testIfOn(tubeLight)` at (3) *satisfies* the parameter lists in both implementations given at (1) and (2), as the reference `tube-Light` can also be assigned to a reference of its superclass `Light`. The *most specific* method, (2), is chosen, resulting in `false` being written on the terminal. The call `client.testIfOn(light)` at (4) satisfies only the parameter list in the implementation given at (1), resulting in `true` being written on the terminal. This is also the case at (5). The object referred to by the argument in the call at runtime is irrelevant; rather, it is the *type* of the argument that is important for overloaded method resolution.

Example 5.19 *Choosing the Most Specific Method (Simple Case)*

Click here to view code image

```
class Light { /* ... */ }

class TubeLight extends Light { /* ... */ }

public class Overload {
  boolean testIfOn(Light aLight)         { return true; }    // (1)
  boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)

  public static void main(String[] args) {

    TubeLight tubeLight = new TubeLight();
    Light     light     = new Light();
    Light     light2    = new TubeLight();

    Overload client = new Overload();
    System.out.println(client.testIfOn(tubeLight)); // (3) ==> method at (2)
    System.out.println(client.testIfOn(light));     // (4) ==> method at (1)
    System.out.println(client.testIfOn(light2));    // (5) ==> method at (2)
  }
}
```

Output from the program:

```
false
true
true
```

The algorithm used by the compiler for the resolution of overloaded methods incorporates the following phases:

- The compiler performs overload resolution without permitting boxing, unboxing, or the use of a variable arity call.
- If phase (1) fails, the compiler performs overload resolution allowing boxing and unboxing, but excluding the use of a variable arity call.
- If phase (2) fails, the compiler performs overload resolution combining a variable arity call, boxing, and unboxing.

Example 5.20 provides some insight into how the compiler determines the most specific overloaded method using these three phases. The example has six overloaded declarations of the method `action()`. The signature of each method is given by the local variable `signature` in each method. The first formal parameter of each method is the *signature of the call* that invoked the method. The printout from each method allows us to see which method call resolved to which method. The `main()` method contains 10 calls, (8) to (17), of the `action()` method. In each call, the first argument is the signature of that method call.

An important point to note is that the compiler chooses a *fixed arity* call over a variable arity call, as seen in the calls from (8) to (12):

Click here to view code image

```
(String) => (String)                                (8) calls (1)
(String, int) => (String, int)                      (9) calls (2)
(String, Integer) => (String, int)                 (10) calls (2)
(String, int, byte) => (String, int, int)          (11) calls (3)
(String, int, int) => (String, int, int)           (12) calls (3)
```

An unboxing conversion (`Integer` to `int`) takes place for the call at (10). A widening primitive conversion (`byte` to `int`) takes place for the call at (11).

Variable arity calls are chosen from (13) to (17):

Click here to view code image

```
(String, int, long) => (String, Number[])          (13) calls (5)
(String, int, int, int) => (String, Integer[])     (14) calls (4)
(String, int, double) => (String, Number[])        (15) calls (5)
(String, int, String) => (String, Object[])        (16) calls (6)
(String, boolean) => (String, Object[])            (17) calls (6)
```

When a variable arity call is chosen, the method determined has the most specific variable arity parameter that is applicable for the actual argument. For example, in the method call at (14), the type `Integer[]` is more specific than either `Number[]` or `Object[]`. Note also the boxing of the elements of the implicitly created array in the calls from (13) to (17).

Example 5.20 *Overloaded Method Resolution*

Click here to view code image

```
import static java.lang.System.out;

class OverloadResolution {

  public void action(String str) {                  // (1)
    String signature = "(String)";
    out.println(str + " => " + signature);
  }

  public void action(String str, int m) {           // (2)
    String signature = "(String, int)";
    out.println(str + " => " + signature);
  }
  public void action(String str, int m, int n) {    // (3)
    String signature = "(String, int, int)";
    out.println(str + " => " + signature);
  }

  public void action(String str, Integer... data) { // (4)
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Number... data) {  // (5)
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Object... data) {  // (6)
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }

  public static void main(String[] args) {
    OverloadResolution ref = new OverloadResolution();
    ref.action("(String)");                                  // (8)  calls (1)
    ref.action("(String, int)",       10);                   // (9)  calls (2)
    ref.action("(String, Integer)",   Integer.valueOf(10));  // (10) calls (2)
    ref.action("(String, int, byte)",     10, (byte)20);     // (11) calls (3)
    ref.action("(String, int, int)",      10,  20);          // (12) calls (3)
    ref.action("(String, int, long)",     10,  20L);         // (13) calls (5)
    ref.action("(String, int, int, int)", 10,  20,  30);     // (14) calls (4)
    ref.action("(String, int, double)",   10,  20.0);        // (15) calls (5)
    ref.action("(String, int, String)",   10,  "what?");     // (16) calls (6)
    ref.action("(String, boolean)",       false);            // (17) calls (6)
  }
}
```

For output from the program, see the explanation for Example 5.20 in the text.

5.11 Reference Casting and the `instanceof` Operator

In this section we explore *type casting of references* and the `instanceof` operator that can be used to perform either *type comparison* or *pattern matching*.

The Cast Operator

The basic form of the type cast expression for reference types has the following syntax:

Click here to view code image

(*destination_type*) *reference_expression* where the *reference expression* evaluates to a reference value of an object of some reference type. A type cast expression checks that the reference value refers to an object whose type is compatible with the *destination type*, meaning that its type is a subtype of the *destination type*. The construct `(`*destination_type*`)` is usually called the *cast operator*. The result of a type cast expression for references is always a reference value of an object. The literal `null` can be cast to any reference type.

The code below illustrates the various scenarios that arise when using the cast operator. In this discussion, it is the type cast expression that is important, not the evaluation of the assignment operator in the declaration statements. At (1), the cast is from the superclass `Object` to the subclass `String`; the code compiles and at runtime this cast is permitted, as the reference `obj` will denote an object of class `String`. At (2), the cast is from the superclass `Object` to the subclass `Integer`; the code compiles, but at runtime this cast results in a `ClassCastException`, since the reference `obj` will denote an object of class `String`, which cannot be converted to an `Integer`. At (3), the cast is from the class `String` to the class `Integer`. As these two classes are unrelated, the compiler flags an error for the cast.

Click here to view code image

```
Object  obj = new String("Cast me!");
String  str = (String) obj;           // (1) Cast from Object to String.
Integer iRef1 = (Integer) obj;        // (2) Cast from Object to Integer, but
                                      //     ClassCastException at runtime.
Integer iRef2 = (Integer) str;        // (3) Compile-time error!
                                      //     Cast between unrelated types.
```

The following conversions can be applied to the operand of a cast operator:

- Both widening and narrowing reference conversions, followed optionally by an unchecked conversion (§11.2, p. 575)
- Both boxing and unboxing conversions

Boxing and unboxing conversions that can occur during casting are illustrated by the following code. Again, it is the type cast expression that is important in this discussion, rather than whether the assignment operator requires one in the declaration statements.

Click here to view code image

```
// (1) Boxing and casting: Number <-- Integer <-- int:
```

```
Number num = (Number) 100;
// (2) Casting, boxing, casting: Object <-- Integer <-- int <-- double:
Object obj = (Object) (int) 10.5;
// (3) Casting, unboxing, casting: double <--- int <-- Integer <-- Object:
double d = (double) (Integer) obj;
```

Note that the resulting object at (1) and (2) is an `Integer`, but the resulting value at (3) is a `double`. The boxing conversions from `int` to `Integer` at (1) and (2) are implicit, and the unboxing conversion from `Integer` to `int` at (3) is also implicit.

Casting to a supertype is always permitted, but redundant, since a supertype reference can always refer to an object of its subtype.

The `instanceof` Type Comparison Operator

The binary `instanceof` operator can be used for comparing *types*. It has the following syntax when used as a type comparison operator (note that the keyword is composed of lowercase letters only):

Click here to view code image

```
reference_expression instanceof destination_type
```

The `instanceof` type comparison operator returns `true` if the left-hand operand (i.e., the reference value that results from the evaluation of *reference expression*) can be a *subtype* of the right-hand operand (*destination type*). It always returns `false` if the left-hand operand is `null`. If the `instanceof` operator returns `true`, the corresponding type cast expression `(`*destination_type*`)` applied to the left-hand operand *reference_expression* will always be valid:

Click here to view code image

```
(destination_type) reference_expression
```

Both the type cast expression and the `instanceof` type comparison operator require a compile-time check and a runtime check. The compile-time check determines whether there is a subtype–supertype relationship between the source and destination types. Given that the type of the *reference expression* is *source type*, the compiler determines whether a reference of *source type* and a reference of *destination type* can refer to objects of a reference type that is a *common subtype* of both *source type* and *destination type* in the type hierarchy. If this is not the case, then obviously there is no relationship between the types, and neither the cast nor the `instanceof` type comparison operator application would be valid.

At runtime, the *reference expression* evaluates to a reference value of an object. The `instanceof` type comparison operator determines whether the type of this object is a subtype of the *destination type*.

With the classes `Light` and `String` as *source type* and *destination type*, respectively, there is no subtype–supertype relationship between *source type* and *destination type*. The compiler would reject casting a reference of type `Light` to type `String` or applying the `instanceof` operator, as shown at (2) and (3) in Example 5.21. References of the classes `Light` and `TubeLight` can refer to objects of the class `TubeLight` (or its subclasses) in the inheritance hierarchy depicted in Figure 5.1, p. 194. Therefore, it makes sense to apply the `instanceof` type comparison operator or to cast a reference of the type `Light` to the type `TubeLight` as shown at (4) and (5), respectively, in Example 5.21.

At runtime, the result of applying the `instanceof` type comparison operator at (4) is `false` because the reference `light1` of the class `Light` will actually denote an object of the subclass `LightBulb`, and this object cannot be denoted by a reference of the peer class `TubeLight`. Applying the cast at (5) results in a `ClassCastException` for the same reason. This is the reason why cast conversions are said to be *unsafe*, as they may throw a `ClassCastException` at runtime. Note that if the result of the `instanceof` type comparison operator is `false`, the cast involving the operands will throw a `ClassCastException`.

In Example 5.21, the result of applying the `instanceof` type comparison operator at (6) is also `false` because the reference `light1` will still denote an object of the class `LightBulb`, whose objects cannot be denoted by a reference of its subclass `SpotLightBulb`. Thus applying the cast at (7) causes a `ClassCastException` to be thrown at runtime.

The situation shown at (8), (9), and (10) illustrates typical usage of the `instanceof` type comparison operator to determine which object a reference is denoting so that it can be cast for the purpose of carrying out some specific action—as we will see later, the `instanceof` pattern match operator provides a better solution for this particular case (p. 274). The reference `light1` of the class `Light` is initialized to an object of the subclass `NeonLight` at (8). The result of the `instanceof` type comparison operator at (9) is `true` because the reference `light1` will denote an object of the subclass `NeonLight`, whose objects can also be denoted by a reference of its superclass `TubeLight`. By the same token, the cast at (10) is valid. If the result of the `instanceof` type comparison operator is `true`, the cast involving the operands will be valid as well.

So far in Example 5.21, the *static* type of the reference in the left-hand operand of the `instanceof` comparison operator has been a *supertype* of the destination type. At (12), the *static* type (`SpotLightBulb`) of the reference in the left-hand operand of the `instanceof` comparison operator is a *subtype* of the destination type (`Light`). Since the *dynamic* type of the object is `SpotLightBulb`, which is a subtype of the destination type `Light`, the `instanceof` comparison operator returns `true`. The reference value of the object can be trivially assigned to a reference of the supertype `Light`, and the cast shown at (13) is actually redundant. This behavior of the `instanceof` comparison operator is allowed because of backward compatibility.

Example 5.21 *The* `instanceof` *Type Comparison and Cast Operators*

Click here to view code image

```
// See Figure 5.1, p. 194, for inheritance hierarchy.
class Light { /* ... */ }
class LightBulb extends Light { /* ... */ }
class SpotLightBulb extends LightBulb { /* ... */ }
class TubeLight extends Light { /* ... */ }
class NeonLight extends TubeLight { /* ... */ }

public class WhoAmI {
  public static void main(String[] args) {
    boolean result1, result2, result3, result4;
    Light light1 = new LightBulb();                    // (1)
    //  String str = (String) light1;                  // (2) Compile-time err
    //  result1 = light1 instanceof String;            // (3) Compile-time err

    result2 = light1 instanceof TubeLight;             // (4) false: peer clas
    //  TubeLight tubeLight1 = (TubeLight) light1;     // (5) ClassCastExcepti

    result3 = light1 instanceof SpotLightBulb;         // (6) false: superclas
    //  SpotLightBulb spotRef = (SpotLightBulb) light1;// (7) ClassCastExcepti

    light1 = new NeonLight();                          // (8)
    if (light1 instanceof TubeLight) {                 // (9) true.
      TubeLight tubeLight2 = (TubeLight) light1;       // (10) OK.
      // Can now use tubeLight2 to access an object of the class NeonLight,
      // but only those members that the object inherits or overrides
      // from the superclass TubeLight.
    }

    SpotLightBulb light2 = new SpotLightBulb();        // (11)
    result4 = light2 instanceof Light;                 // (12) true.
    Light light = (Light) light2;                      // (13) OK. Redundant c
  }
}
```

As we have seen, the `instanceof` type comparison operator effectively determines whether the reference value in the reference on the left-hand side refers to an object whose class is a subtype of the type specified on the right-hand side. At runtime, it is the type of the actual object denoted by the reference on the left-hand side that is compared with the type specified on the right-hand side. In other words, what matters at runtime is the type of the actual object denoted by the reference, not the declared or static type of the reference.

Example 5.22 provides more examples of the `instanceof` type comparison operator. It is instructive to go through the print statements and understand the results printed by the program. The following facts should be noted:

- The literal `null` is not an instance of any reference type, as shown in the print statements at (1), (2), and (16).
- An instance of a superclass is not an instance of its subclass, as shown in the print statement at (4).
- An instance of a class is not an instance of a totally unrelated class, as shown in the print statement at (10).
- An instance of a class is not an instance of an interface type that the class does not implement, as shown in the print statement at (6).
- Any array of a non-primitive type is an instance of both `Object` and `Object[]` types, as shown in the print statements at (14) and (15), respectively.

Example 5.22 *Using the* `instanceof` *Type Comparison Operator*

Click here to view code image

```
// See Figure 5.4, p. 245, for inheritance hierarchy.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

Click here to view code image

```
public class Identification {
  public static void main(String[] args) {
    Object obj = new Object();
    Stack stack = new Stack(10);
    SafeStack safeStack = new SafeStack(5);
    IStack iStack;

    String strFormat = "(%d)  %-25s instance of %-25s: %s%n";
    System.out.printf(strFormat, 1,
        null, Object.class, null instanceof Object);      // Always false.
    System.out.printf(strFormat, 2,
        null, IStack.class, null instanceof IStack);      // Always false.
        System.out.printf(strFormat, 3, stack.getClass(), Object.class,
            stack instanceof Object);      // true: instance of subclass of Object.
        System.out.printf(strFormat, 4, obj.getClass(), Stack.class,
            obj instanceof Stack);         // false: Object not subtype of Stack.
        System.out.printf(strFormat, 5, stack.getClass(), Stack.class,
            stack instanceof Stack);       // true: instance of Stack.
        System.out.printf(strFormat, 6, obj.getClass(), IStack.class,
            obj instanceof IStack);        // false: Object does not implement ISta
        System.out.printf(strFormat, 7, safeStack.getClass(), IStack.class,
            safeStack instanceof IStack); // true: SafeStack implements IStack.

        obj = stack;           // No cast required: assigning subclass to supercla
        System.out.printf(strFormat, 8, obj.getClass(), Stack.class,
            obj instanceof Stack);         // true: instance of Stack.
        System.out.printf(strFormat, 9, obj.getClass(), IStack.class,
            obj instanceof IStack);        // true: Stack implements IStack.
        System.out.printf(strFormat, 10, obj.getClass(), String.class,
            obj instanceof String);        // false: no relationship.
        iStack = (IStack) obj; // Cast required: assigning superclass to subclass.
        System.out.printf(strFormat, 11, iStack.getClass(), Object.class,
            iStack instanceof Object);        // true: instance of subclass of Obje
        System.out.printf(strFormat, 12, iStack.getClass(), Stack.class,
            iStack instanceof Stack);         // true: instance of Stack.

        String[] strArray = new String[10];
    //  System.out.printf(strFormat, 13, strArray.getClass(), String.class,
    //      strArray instanceof String);      // Compile-time error: no relationshi
        System.out.printf(strFormat, 14, strArray.getClass(), Object.class,

            strArray instanceof Object);      // true: array subclass of Object.
        System.out.printf(strFormat, 15, strArray.getClass(), Object[].class,
            strArray instanceof Object[]);    // true: array subclass of Object[].
        System.out.printf(strFormat, 16, strArray[0], Object.class,
            strArray[0] instanceof Object);   // false: strArray[0] is null.
        System.out.printf(strFormat, 17, strArray.getClass(), String[].class,
            strArray instanceof String[]);    // true: array of String.

        strArray[0] = "Amoeba strip";
        System.out.printf(strFormat, 18, strArray[0].getClass(), String.class,
            strArray[0] instanceof String);   // true: strArray[0] instance of Stri
      }
    }
```

Output from the program:

Click here to view code image

```
(1)  null                      instance of class java.lang.Object   : false
(2)  null                      instance of interface IStack         : false
(3)  class Stack               instance of class java.lang.Object   : true
(4)  class java.lang.Object    instance of class Stack              : false
(5)  class Stack               instance of class Stack              : true
(6)  class java.lang.Object    instance of interface IStack         : false
(7)  class SafeStack           instance of interface IStack         : true
(8)  class Stack               instance of class Stack              : true
(9)  class Stack               instance of interface IStack         : true
(10)  class Stack               instance of class java.lang.String   : false
(11)  class Stack               instance of class java.lang.Object   : true
(12)  class Stack               instance of class Stack              : true
(14)  class [Ljava.lang.String; instance of class java.lang.Object   : true
(15)  class [Ljava.lang.String; instance of class [Ljava.lang.Object;: true
(16)  null                      instance of class java.lang.Object   : false
(17)  class [Ljava.lang.String; instance of class [Ljava.lang.String;: true
(18)  class java.lang.String    instance of class java.lang.String   : true
```

The `instanceof` Pattern Match Operator

The `instanceof` operator can also be used for *pattern matching*, as explained below. It has the following syntax when used as a *pattern match operator*:

Note that the syntax of the `instanceof` pattern match operator augments the syntax of the `instanceof` type comparison operator with a *pattern variable*. The *type pattern* comprises a *local reference variable declaration*. The `instanceof` pattern match operator essentially combines the `instanceof` type comparison operator with a type cast and initialization of a *pattern variable*.

As with the `instanceof` type comparison operator, a compile-time check determines whether there is a subtype–supertype relationship between the static type of the left-hand operand and the *destination type*. However, as opposed to the `instanceof` type comparison operator, the compiler flags an error if the *destination type* is *not* a *subtype* of the *static* type of the left-hand operand.

At runtime, if the reference value of the object resulting from the evaluation of the *reference expression* can be *cast* to the *destination type* (i.e., the dynamic type of the resulting object is a *subtype* of the *destination type*), the object is said to *match* the *type pattern*. Only in this case does the `instanceof` pattern match operator return `true` and, as a side effect, *the reference value of the object is cast to the destination type and assigned to the pattern variable*. The *pattern variable* is never created or initialized if the `instanceof` pattern match operator returns `false`. It always returns `false` if the left-hand operand is `null`.

We assume the following types in the discussion below. See Example 5.10, p. 241, for type declarations.

Click here to view code image

```
interface IStack                    (defines methods push() and pop())
interface ISafeStack extends IStack (with boolean methods isEmpty() and isFull
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

The `instanceof`-and-cast idiom is commonly used to elicit the subtype-specific behavior of an object referenced by a supertype reference. The code below at (1) and (2) is equivalent. However, this idiom can be expressed compactly with the `instanceof` pattern match operator as it type checks the object and casts its reference value that is assigned to the pattern variable. When this operator returns `true`, it is said to *introduce* a *pattern variable* into a well-defined scope, which in this case is the `if` block.

Click here to view code image

```
IStack stack = new SafeStack(20); // Supertype reference denotes subtype objec

// Using the instanceof type comparison operator. (1)
if (stack instanceof SafeStack) {                 // Correct subtype?
  SafeStack safestack = (SafeStack) stack;        // Cast to subtype.
  System.out.println(safestack.isFull());         // Call subtype-specific met
}

// Using the instanceof pattern match operator.   (2)
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());
}
```

In contrast to the `instanceof` type comparison operator, the destination type in the `instanceof` pattern match operator *must* be a subtype of the static type of the left-hand operand. In other words, the reference value of the object is always cast to a subtype by the operator. In the code below, the compiler complains that the expression type (`ISafeStack`) cannot be a subtype of the pattern types `ISafeStack` and `IStack`, respectively, at (1) and (2), in the context of the `instanceof` pattern match operator.

Click here to view code image

```
ISafeStack safestack = new SafeStack(20);
if (safestack instanceof ISafeStack stack) {     // (1) Compile-time error!
    stack.push("Hi");
}
if (safestack instanceof IStack stack) {         // (2) Compile-time error!
    stack.push("Howdy");
}
```

A pattern variable *cannot* shadow another local variable by the same name—that is, it cannot be redeclared in a pattern if the local variable of the same name is still in scope.

Click here to view code image

```
IStack stack = new SafeStack(20);
SafeStack safestack = new SafeStack(5);        // Local variable safestack

if (stack instanceof SafeStack safestack) {    // Error: safestack redeclared.
  System.out.println(safestack.isFull());
}
```

A pattern variable introduced by the `instanceof` pattern match operator can shadow a *field* of the same name in the class.

Click here to view code image

```
// Field declaration:
```

```
private static IStack myStack = new Stack(10);

...
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack myStack) {    // Local variable introduced.
  System.out.println(myStack.isFull());      // Shadows field reference.
}
myStack.push("Hello");                       // Field reference.
```

A pattern variable is not `final` in its scope unless it is declared `final`—in other words, the pattern variable can be declared with the modifier `final`, as shown at (1). Changing the value of a `final` pattern variable is flagged as a compile-time error, as shown at (2).

Click here to view code image

```
IStack stack = new SafeStack(
if (stack instanceof final SafeStack safestack) {  // (1) final pattern variab
  safestack = new SafeStack(100);                  // (2) Compile-time error!
  System.out.println(safestack.isFull());
}
```

Scope of Pattern Variables

We first examine the scope of a pattern variable in an `if-else` statement. If the `instanceof` pattern match operator returns `true` in the conditional of an `if` statement, the pattern variable is introduced and its scope is the `if` block. Not surprisingly, this is also the case for the `if-else` statement. The pattern variable is *not* accessible in the `else` block.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());   // Pattern variable in scope.
} else {

  System.out.println(safestack.isEmpty());  // Compile-time error!
}
```

It is important to keep in mind that the `instanceof` pattern match operator introduces a pattern variable if and only if it returns `true`. In the `if-else` statement below, the `if` block is only executed if the conditional is `true`—that is, if the `instanceof` pattern match operator returns `false`. In that case, no pattern variable is introduced, and thus no pattern variable is ever accessible in the `if` block. However, if the conditional is `false`, the `instanceof` pattern match operator must be `true`, and thus introduces a pattern variable that is guaranteed to be accessible in the `else` block.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) { // Logical complement operator
  System.out.println(safestack.isFull());      // Compile-time error.
} else {
  System.out.println(safestack.isFull());      // Pattern variable in scope.
}
```

In fact, if the `if` block does *not complete normally* (e.g., by executing a `return`, a `break`, or a `continue` statement) as shown at (1), the pattern variable introduced is accessible in the `else` block and in code after the `if-else` statement.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) {
  System.out.println("No safestack here");

  return;                                     // (1) Does not complete normall
} else {
  System.out.println(safestack.isFull());     // Pattern variable in scope.
}
System.out.println(safestack.isEmpty());      // Pattern variable still in sco
```

The `instanceof` pattern match operator can also introduce a pattern variable in certain boolean expressions. The conditional in the `if` statement below uses the conditional-AND operator (`&&`). The short-circuit evaluation of the `&&` operator ensures that the right-hand operand is only executed if the left-hand operand evaluates to `true`, thereby introducing the pattern variable that is then in scope in the right-hand operand. The pattern variable will be in scope in the `if` block if the conditional evaluates to `true`—that is, both operands of the `&&` operator return `true`. Applying the logical complement (`!`) operator to the conditional expression below works the same way as we have seen earlier with the `if-else` statement.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack && safestack.isFull()) {
  System.out.println("safestack is full");

  Object obj = safestack.pop();
}
```

The conditional-OR operator (`||`) does *not* introduce a pattern variable in the `if` block. Because of short-circuit evaluation of the `||` operator, the right-side operand of the conditional is only evaluated if the left-side operand is `false`, but then no pattern variable has been introduced in the boolean expression by the `instanceof` pattern match operator.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack || safestack.isFull()) { // Compile-t
                                                                     error.
  System.out.println("safestack is full");
  Object obj = safestack.pop();                             // Compile-time er
}
```

Using the same pattern variable in `instanceof` expressions below results in a compile-time error, as this is analogous to redeclaring a local variable.

Click here to view code image

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack &&  // Compile-time error!
    stack instanceof ISafeStack safestack) { // Duplicate variable safestack
  System.out.println(safestack);
}
```

The `instanceof` pattern match operator can be used in the conditional (ternary) operator, and it's semantics are analogous to those of the `if-else` statement:

Click here to view code image

```
IStack stack = new SafeStack(20);
boolean result = stack instanceof SafeStack safestack ? safestack.isEmpty()
                                                      : false;
```

The `instanceof` pattern match operator can also introduce a pattern variable in loops—for example, in `for` and `while` statements. This allows conditional processing of objects where loop termination is controlled by the `boolean` value of the `instanceof` pattern match operator and the pattern variable referring to the next object for processing.

Using the `instanceof` pattern match operator makes the code concise and safe, as it combines three tasks into a single operation: subtype checking, reference casting, and assignment to a local variable.

5.12 Polymorphism

A supertype reference can denote an object of its subtypes—conversely, a subtype object can act as an object of its supertypes.

This is because there is an *is-a* relationship between a subtype and its supertypes.

Since a supertype reference can denote objects of different types at different times during execution, a supertype reference is said to exhibit *polymorphic* behavior (meaning *has many forms*). A subtype object also exhibits polymorphic behavior, since it can act as any object of its supertypes.

When a non-`private` instance method is called, the method definition executed is determined by *dynamic method lookup*, based on *the type of the object* denoted by the reference at runtime. Dynamic method lookup (also known as *late binding*, *dynamic binding*, and *virtual method invocation*) is the process of determining which method definition a method call signature denotes at runtime. It is performed starting in the class of the object on which the method is invoked, and moves up the inheritance hierarchy to find the supertype with the appropriate method definition.

For *a call to an overridden method using a supertype reference*, dynamic method lookup can determine a different method definition to execute at different times depending on the type of the object that is denoted by the supertype reference during execution. The overridden method is said to exhibit polymorphic behavior. Exploiting this polymorphic behavior is illustrated next.

The inheritance hierarchy depicted in Figure 5.7 is implemented in Example 5.23. Note that both the `draw()` and `area()` methods are overridden in Figure 5.7. The `abstract` method `draw()` in the `IDrawable` interface at (1) is implemented in all subclasses of the `abstract` class `Shape`. It is also implemented by the class `Graph`. The invocation of the `draw()` method in the three loops at (3), (4), and (6) in Example 5.23 relies on the polymorphic behavior of references and dynamic method lookup. The array `drawables` holds `IDrawable` references that can be assigned the reference value of an object whose class implements the `IDrawable` interface: a `Square`, a `Circle`, a `Rectangle`, and a `Graph`, as shown at (2). At runtime, dynamic lookup determines the `draw()` method implementation that will execute, based on the type of the object denoted by each element in the array.

Figure 5.7 *Type Hierarchy to Illustrate Polymorphism*

Example 5.23 *Using Polymorphism*

Click here to view code image

```
// File: PolymorphRefs.java
interface IDrawable {                                                     // (
  void draw();
}
//____________________________________________________________________________
abstract class Shape implements IDrawable {
  abstract public void area();
}
//____________________________________________________________________________
class Circle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Circle."); }
  @Override public void area() {
    System.out.println("Computing area of a Circle.");
  }
}
//____________________________________________________________________________
class Rectangle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Rectangle."); }
  @Override public void area() {
    System.out.println("Computing area of a Rectangle.");
  }
}
//____________________________________________________________________________
class Square extends Rectangle {
  @Override public void draw() { System.out.println("Drawing a Square."); }
  @Override public void area() {
    System.out.println("Computing area of a Square.");
  }
}
//____________________________________________________________________________
class Graph implements IDrawable {
  @Override public void draw() { System.out.println("Drawing a Graph."); }
}
//____________________________________________________________________________
public class PolymorphRefs {
  public static void main(String[] args) {
    IDrawable[] drawables
        = {new Square(), new Circle(), new Rectangle(), new Graph()};     // (

    System.out.println("Draw drawables:");
    for (IDrawable drawable : drawables) {                                // (
      drawable.draw();
    }

    System.out.println("Only draw shapes:");
    for (IDrawable drawable : drawables) {                                // (
      if (drawable instanceof Shape) {                                    // (
        drawable.draw();
      }
    }

    System.out.println("Only compute area of shapes:");
    for (IDrawable drawable : drawables) {                                // (
      if (drawable instanceof Shape shape) {                              // (
        shape.area();                                                     // (
      }
    }
  }
}
```

Output from the program:

Click here to view code image

```
Draw drawables:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Drawing a Graph.
Only draw shapes:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Only compute area of shapes:
Computing area of a Square.
Computing area of a Circle.
Computing area of a Rectangle.
```

In the `for(:)` loop at (3) in Example 5.23, depending on the type of the object denoted by the loop variable `drawable`, the call to the `draw()` method will result in the `draw()` method of this object to be executed.

The `instanceof` type comparison operator returns `true` if the reference specified as its left-hand operand at runtime denotes an object whose type is the same as or is a subtype of the reference type specified in its right-hand operand. If the types of the operands are unrelated, the compiler issues an error.

Click here to view code image

```
IDrawable d1 = new Rectangle();
System.out.println(d1 instanceof IDrawable); // true. Rectangle is an IDrawabl
System.out.println(d1 instanceof Shape);     // true. Rectangle is a Shape.
System.out.println(d1 instanceof Rectangle); // true. Rectangle is a Rectangle
System.out.println(d1 instanceof Circle);    // false. Rectangle is not a Circ
System.out.println(d1 instanceof Graph);     // false. Rectangle is not a Grap
// System.out.println(d1 instanceof String); // Unrelated. Compile-time error.
```

In Example 5.23, the `abstract` method `area()` of the `abstract` class `Shape` is implemented by all subclasses of the superclass `Shape`. The code below does not compile, as the compiler determines that the method `area()` is not defined by the interface `IDrawable`.

Click here to view code image

```
IDrawable d2 = new Square();  // Subtype object denoted by supertype reference
d2.area();        // Method not defined for type IDrawable. Compile-time error
```

In order to elicit subtype-specific behavior in an object that is denoted by a super-type reference, the reference must be *cast* to the subtype. The cast is specified as `(`*reference_type*`)`. The cast in this case applies a narrowing reference conversion from a supertype to a subtype. The cast appeases the compiler. At runtime, the type of the object still determines which `area()` method will be executed.

Click here to view code image

```
((Square) d2).area();         // Prints "Computing area of a Square."
((Shape) d2).area();          // Prints "Computing area of a Square."
```

The code above executes normally because the reference `d2` at runtime denotes an object whose type defines the appropriate `area()` method. The code below shows that the cast alone is not enough to guarantee that the execution will proceed normally, even though there was no compile-time error. The `Graph` object denoted by the reference `d3` at runtime does not define the `area()` method.

Click here to view code image

```
IDrawable d3 = new Graph();       // No compile-time error.
((Shape) d3).area();              // Throws a ClassCastException!
```

Guaranteeing the correct subtype and casting to a subtype reference safely can be accomplished using the `instanceof` pattern match operator.

Click here to view code image

```
if (d2 instanceof Shape shape) {           // true
  shape.area();                            // Prints "Computing area of a Squa
} else {
  System.out.println(d2.getClass().getName() + " is not a Shape." );
}

if (d3 instanceof Shape shape) {           // false
  shape.area();
} else {
  System.out.println(d3.getClass().getName() +
                     " is not a Shape." ); // Prints "Graph is not a Shape."
}
```

In the `for(:)` loop at (4) in Example 5.23, we are only interested in drawing objects from the `drawables` array that are of type `Shape`. The binary `instanceof` type comparison operator in the conditional of the `if` statement at (5) is used to determine whether the type of an object in the `drawables` array is of type `Shape`. The `draw()` method is only called on objects that satisfy this condition.

The `for(:)` loop at (6) in Example 5.23 uses the `instanceof` pattern match operator at (7) to select those `IDrawable` objects whose type is `Shape`, and invoke the `area()` method on them.

A `private` instance method does not exhibit polymorphic behavior. A call to such a method can occur only within the class and gets bound to the `private` method implementation at compile time.

Overloaded instance methods do not exhibit polymorphic behavior, as their calls are bound at compile time, unless an overloaded method is also overridden and invoked by a supertype reference.

Static methods also do not exhibit polymorphic behavior, as these methods do not belong to objects.

Polymorphism is achieved through inheritance and interface implementation. Code relying on polymorphic behavior will still work without any change if new subclasses or new classes implementing the interface are added. If no obvious *is-a* relationship is present, polymorphism is best achieved by using aggregation with interface implementation.

Polymorphism and dynamic method lookup form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamically changing relationships between objects at runtime.

Review Questions

5.22 Given the following class and reference declarations, what can be said about the statement `y = (Sub) x`?

Click here to view code image

```
// Class declarations:
class Super {}
class Sub extends Super {}

// Reference declarations:
  Super x = null;
  Sub y = null;
```

Select the one correct answer.

a. It is illegal at compile time.

b. It is legal at compile time, but it might throw a `ClassCastException` at runtime.

c. It is definitely legal at runtime, but the cast operator `(Sub)` is not strictly needed.

d. It is definitely legal at runtime, and the cast operator `(Sub)` is needed.

5.23 Given the following code:

Click here to view code image

```
class A { }
class B extends A { }
class C extends B { }
```

Which `boolean` expression is `true` only when the reference `o` refers to an object of class `B`, and not to an object of class `A` or class `C`?

Select the one correct answer.

a. `(o instanceof B) && (!(o instanceof A))`

b. `(o instanceof B) && (!(o instanceof C))`

c. `!((o instanceof A) || (o instanceof B))`

d. `(o instanceof B)`

e. `(o instanceof B) && !((o instanceof A) || (o instanceof C))`

5.24 What will the following program print when run?

Click here to view code image

```
interface I{}
interface J{}
class C implements I {}
class D extends C implements J {}

public class RQ07A100 {
  public static void main(String[] args) {
    I x = new D();
    if (x instanceof I) System.out.print("I");
    if (x instanceof J) System.out.print("J");
    if (x instanceof C) System.out.print("C");
    if (x instanceof D) System.out.print("D");
    System.out.println();
  }
}
```

Select the one correct answer.

a. The program will not print any letters.

b. `ICD`

c. `IJD`

d. `IJCD`

e. `ID`

5.25 What will be the result of compiling and running the following program?

Click here to view code image

```
public class RQ800A20 {
  static void compute(int... ia) {                                    // (1)
    System.out.print("|");
    for(int i : ia) {
      System.out.print(i + "|");
    }
    System.out.println();
  }
  static void compute(int[] ia1, int... ia2) {                        // (2)
    compute(ia1);
    compute(ia2);
  }
  static void compute(int[] ia1, int[]... ia2d) {                     // (3)
    for(int[] ia : ia2d) {
      compute(ia);
    }
  }
  public static void main(String[] args) {
    compute(new int[] {10, 11}, new int[] {12, 13, 14});             // (4)
    compute(15, 16);                                                 // (5)
    compute(new int[] {17, 18}, new int[][] {{19}, {20}});           // (6)
    compute(null, new int[][] {{21}, {22}});                         // (7)
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of errors in one or more calls to the `compute()` method.

b. The program compiles, but it throws a `NullPointerException` when run.

c. The program compiles and prints:

```
|10|11|
|12|13|14|
|15|16|
|19|
|20|
|21|
|22|
```

d. The program compiles and prints:

```
|12|13|14|
|15|16|
|10|11|
|19|
|20|
|21|
|22|
```

5.26 Given the following code:

Click here to view code image

```
public class RQ43 {
  public static void main(String[] args) {
    Object x = "acme";
    if (x instanceof String s && s.length() > 5) {
      x = s.equals("acme") ? "1" : "2";
    } else {
      x = s.equals("acme") ? "3" : "4";
    }
    System.out.println(x);
  }
}
```

What is the result?

Select the one correct answer.

a. `1`
b. `2`
c. `3`
d. `4`

- **e.** The program will throw an exception at runtime.
- **f.** The program will fail to compile.

5.27 Which option will compile without errors?

Select the one correct answer.

- **a.** Click here to view code image

```
Integer x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
```

- **b.** Click here to view code image

```
var x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

- **c.** Click here to view code image

```
var x = Integer.valueOf(42);
if (x instanceof String s) {
    System.out.print(s.toUpperCase());
}
```

- **d.** Click here to view code image

```
Number x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

- **e.** Click here to view code image

```
Integer x = Integer.valueOf(42);
if (x instanceof Number s) {
    System.out.print(s.intValue());
}
```

5.28 Which of the following statements are true? Select the two correct answers.

- **a.** The `instanceof` pattern match operator does not throw a `NullPointerException` if its left operand is `null`.
- **b.** The `instanceof` pattern match operator throws a `NullPointerException` if its left operand is `null`.
- **c.** A pattern variable is only introduced when the `instanceof` pattern match operator returns `true`.
- **d.** A pattern variable is only introduced when the `instanceof` pattern match operator returns `false`.

5.29 What will be the result of compiling and running the following program?

Click here to view code image

```
class A           { int f() { return 0; } }
class B extends A { int f() { return 1; } }
class C extends B { int f() { return 2; } }

public class Polymorphism {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.f());
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw a `ClassCastException` at runtime.

c. The program will compile and print `0` when run.

d. The program will compile and print `1` when run.

e. The program will compile and print `2` when run.

5.30 What will be the result of compiling and running the following program?

Click here to view code image

```
class A {
  private int f() { return 0; }
  public int g() { return 3; }
}
class B extends A {
  private int f() { return 1; }
  public int g() { return f(); }
}
class C extends B {
  public int f() { return 2; }
}

public class Polymorphism2 {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.g());
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile and print `0` when run.

c. The program will compile and print `1` when run.

d. The program will compile and print `2` when run.

e. The program will compile and print `3` when run.

5.31 Which of the following statements about the following program are true?

Click here to view code image

```
public interface HeavenlyBody { String describe(); }

class Star {
  String starName;
  public String describe() { return "star " + starName; }
}

class Planet extends Star {
  String name;
  public String describe() {
    return "planet " + name + " orbiting star " + starName;
  }
}
```

Select the three correct answers:

a. The code will fail to compile.

b. The code defines a `Planet` *is-a* `Star` relationship.

c. The code will fail to compile if the name `starName` is replaced with the name `bodyName` throughout the declaration of the `Star` class.

d. The code will fail to compile if the name `starName` is replaced with the name `name` throughout the declaration of the `Star` class.

e. An instance of `Planet` is a valid instance of `HeavenlyBody`.

f. The code defines a `Planet` *has-a* `Star` relationship.

5.13 Enum Types

An *enum type* is a special-purpose class that defines *a finite set of symbolic names and their values*. These symbolic names are usually called *enum constants* or *named constants*. An enum type is also synonymously referred to as an *enum class*.

Before the introduction of enum types in the Java programming language, such constants were typically declared as `final`, `static` variables in a class (or an interface) declaration:

Click here to view code image

```
public class MachineState {
  public static final int BUSY = 1;
  public static final int IDLE = 0;
  public static final int BLOCKED = -1;
}
```

Such constants are not type-safe, as *any* `int` value can be used where we need to use a constant declared in the `MachineState` class. Such a constant must be qualified by the class (or interface) name, unless the class is extended (or the interface is implemented). When such a constant is printed, only its value (e.g., `0`), and not its name (e.g., `IDLE`) is printed. A constant also needs recompiling if its value is changed, as the values of such constants are compiled into the client code.

An enum type in Java is a special kind of class that is much more powerful than the approach outlined above for defining named constants.

Declaring Type-Safe Enums

The canonical form of declaring an enum type is shown below.

Click here to view code image

```
access_modifier  enum enum_type_name    // Enum header
                                         // Enum body
  EC1, EC2, ..., ECk                    // Enum constants
}
```

The following declaration is an example of an enum type:

Click here to view code image

```
public enum MachineState                  // Enum header
{                                         // Enum body
  BUSY, IDLE, BLOCKED                     // Enum constants
}
```

The keyword `enum` is used to declare an enum type, as opposed to the keyword `class` for a class declaration. The basic notation requires the *enum type name* in the enum header, and *a comma-separated list of enum constants* (EC_1*, EC2, ..., ECk*) can be specified in the enum body. In the example enum declaration, the name of the enum type is `MachineState`. It defines three enum constants with explicit names: `BUSY`, `IDLE`, and `BLOCKED`. An enum constant can be any legal Java identifier, but the convention is to use uppercase letters in the name.

Essentially, an enum declaration defines a *reference type* that has a *finite number of permissible values* referenced by the enum constants, and the compiler ensures they are used in a type-safe manner.

Analogous to a top-level class, a top-level enum type can be declared with either `public` or package accessibility. However, an enum type declared as a `static` member of a reference type can be declared with any accessibility.

As we shall see later, other member declarations can be specified in the body of an enum type. If this is the case, the enum constant list must be terminated by a semicolon (`;`). Analogous to a class declaration, an enum type is compiled to Java byte-code that is placed in a separate class file.

The Java SE Platform API contains numerous enum types. We mention two enum types here: `java.time.Month` and `java.time.DayOfWeek`. As we would expect, the `Month` enum type represents the months from `JANUARY` to `DECEMBER`, and the `DayOfWeek` enum type represents the days of the week from `MONDAY` to `SUNDAY`. Examples of their usage can be found in §17.2, p. 1027.

Some additional examples of enum types are given below.

Click here to view code image

```
public enum MarchingOrders { LEFT, RIGHT }

public enum TrafficLightState { RED, YELLOW, GREEN; }

enum MealType { BREAKFAST, LUNCH, DINNER }
```

Using Type-Safe Enums

Example 5.24 illustrates using enum constants. An enum type is essentially used as any other reference type, and the restrictions are noted later in this section. Enum constants are actually `final`, `static` variables of the enum type, and they are implicitly initialized with instances of the enum type when the enum type is loaded at runtime. Since the enum constants are `static` members, they can be accessed using the name of the enum type—analogous to accessing `static` members in a class.

Example 5.24 shows a machine client that uses a machine whose state is an enum constant. In this example, we see that an enum constant can be passed as an argument, as shown at (1), and we can declare references whose type is an enum type, as shown at (3), but we *cannot* create new constants (i.e., objects) of the enum type `MachineState`. An attempt to do so at (5) results in a compile-time error.

The text representation of an enum constant is its name, as shown at (4). Note that it is not possible to pass a value of a type other than a `MachineState` enum constant in the call to the method `setState()` of the `Machine` class, as shown at (2).

Example 5.24 *Using Enums*

Click here to view code image

```
// File: MachineState.java
public enum MachineState { BUSY, IDLE, BLOCKED }
```

Click here to view code image

```
// File: Machine.java
public class Machine {

  private MachineState state;

  public void setState(MachineState state) { this.state = state; }
  public MachineState getState()           { return this.state; }
}
```

Click here to view code image

```
// File: MachineClient.java
public class MachineClient {
  public static void main(String[] args) {

    Machine machine = new Machine();
    machine.setState(MachineState.IDLE);            // (1) Passed as a value.
    // machine.setState(1);                         // (2) Compile error!
    MachineState state = machine.getState();        // (3) Declaring a referen
    System.out.println(
        "Current machine state: " + state           // (4) Printing the enum n
    );

    // MachineState newState = new MachineState();  // (5) Compile error!

    System.out.println("All machine states:");
    for (MachineState ms : MachineState.values()) { // (6) Traversing over enu
      System.out.println(ms + ":" + ms.ordinal());  //     contants.
    }

    System.out.println("Comparison:");
    MachineState state1 = MachineState.BUSY;
    MachineState state2 = state1;
    MachineState state3 = MachineState.BLOCKED;

    System.out.println(state1 + " == " + state2 + ": " +
                       (state1 == state2));                           // (7)
    System.out.println(state1 + " is equal to " + state2 + ": " +
                       (state1.equals(state2)));                      // (8)
    System.out.println(state1 + " is less than " + state3 + ": " +
                       (state1.compareTo(state3) < 0));               // (9)
  }
}
```

Output from the program:

Click here to view code image

```
Current machine state: IDLE
All machine states:
BUSY:0
IDLE:1
BLOCKED:2
Comparison:
BUSY == BUSY: true
BUSY is equal to BUSY: true
BUSY is less than BLOCKED: true
```

Declaring Enum Constructors and Members

An enum type can declare constructors and other members as in an ordinary class, but the enum constants must be declared before any other declarations (see the declaration of the enum type `Meal` in Example 5.25). The list of enum constants must be terminated by a semicolon (`;`) if followed by any constructor or member declaration. Each enum constant name can be followed by an argument list that is passed to the constructor of the enum type having the matching parameter signature.

In Example 5.25, the enum type `Meal` contains a constructor declaration at (2) with the following signature:

```
Meal(int, int)
```

Each enum constant is specified with an argument list with the signature `(int, int)` that matches the non-zero argument constructor signature. The enum constant list is also terminated by a semicolon, as the enum declaration contains other members: two fields for the meal time at (3), and three instance methods to retrieve meal time information at (4).

When the enum type is loaded at runtime, the constructor is run for each enum constant, passing the argument values specified for the enum constant. For the `Meal` enum type, three objects are created that are initialized with the specified argument values, and are referenced by the three enum constant names, respectively. Note that each enum constant is a `final`, `static` reference that stores the reference value of an object of the enum type, and methods of the enum type can be called on this object by using the enum constant name. This is illustrated at (5) in Example 5.25 by calling methods on the object referenced by the enum constant `Meal.BREAKFAST`.

A default constructor is created if no constructors are provided for the enum type, analogous to a class. As mentioned earlier, an enum type cannot be instantiated using the `new` operator. The constructors cannot be called explicitly. Thus the only access modifier allowed for a constructor is `private`, as a constructor is understood to be implicitly declared `private` if no access modifier is specified.

Static initializer blocks can also be declared in an enum type, analogous to those in a class (§10.7, p. 545).

Example 5.25 *Declaring Enum Constructors and Members*

Click here to view code image

```
// File: Meal.java
public enum Meal {
  BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);                  // (1)

  // Non-zero argument constructor                                  (2)
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Fields for the meal time:                                      (3)
  private int hh;
  private int mm;

  // Instance methods:                                              (4)
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                                  // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}
```

Click here to view code image

```
// File: MealAdministrator.java
public class MealAdministrator {
  public static void main(String[] args) {

    System.out.printf(                                            // (5)
        "Please note that no eggs will be served at %s, %s.%n",
        Meal.BREAKFAST, Meal.BREAKFAST.getTimeString()
    );

    System.out.println("Meal times are as follows:");
    Meal[] meals = Meal.values();                                 // (6)
    for (Meal meal : meals) {                                     // (7)
      System.out.printf("%s served at %s%n", meal, meal.getTimeString());
    }

    Meal formalDinner = Meal.valueOf("DINNER");                   // (8)
    System.out.printf("Formal dress is required for %s at %s.%n",
        formalDinner, formalDinner.getTimeString()
    );
  }
}
```

Output from the program:

Click here to view code image

```
Please note that no eggs will be served at BREAKFAST, 07:30.
Meal times are as follows:
BREAKFAST served at 07:30
LUNCH served at 12:15
DINNER served at 19:45
Formal dress is required for DINNER at 19:45.
```

Implicit Static Methods for Enum Types

All enum types implicitly have the following `static` methods, and methods with these names *cannot* be declared in an enum type declaration:

Click here to view code image

```
static EnumTypeName[] values()
```

Returns an array containing the enum constants of this enum type, *in the order they are specified.*

Click here to view code image

```
static EnumTypeName valueOf(String name)
```

Returns the enum constant with the specified `name`. An `IllegalArgumentException` is thrown if the specified `name` does not match the name of an enum constant. The specified `name` is *not* qualified with the enum type name.

The `static` method `values()` is called at (6) in Example 5.25 to create an array of enum constants. This array is traversed in the `for(:)` loop at (7), printing the information about each meal.

The `static` method `valueOf()` is called at (8) in Example 5.25 to retrieve the enum constant that has the specified name `"DINNER"`. A print statement is used to print the information about the meal denoted by this enum constant.

Inherited Methods from the `java.lang.Enum` Class

All enum types are subtypes of the `java.lang.Enum` class which implements the default behavior. All enum types are comparable (§14.4, p. 761) and serializable (§20.5, p. 1261).

All enum types inherit the following selected `final` methods from the `java.lang.Enum` class, and these methods therefore *cannot* be overridden by an enum type:

```
final int compareTo(E o)
```

The *natural order* of the enum constants in an enum type is according to their *ordinal values* (see the `ordinal()` method below). The `compareTo()` method in the `Comparable` interface is discussed in §14.4, p. 761.

Click here to view code image

```
final boolean equals(Object other)
```

> Returns `true` if the specified object is equal to this enum constant (§14.2, p. 744).
>
> `final int hashCode()`
>
> Returns a hash code for this enum constant (§14.3, p. 753).
>
> `final String name()`
>
> Returns the name of this enum constant, exactly as it is declared in its enum declaration.
>
> `final int ordinal()`
>
> Returns the *ordinal value* of this enum constant (i.e., its position in its enum type declaration). The first enum constant is assigned an ordinal value of zero. If the ordinal value of an enum constant is less than the ordinal value of another enum constant of the same enum type, the former occurs before the latter in the enum type declaration.

Note that the equality test implemented by the `equals()` method is based on reference equality (==) of the enum constants, not on object value equality. Comparing two enum references for equality meanss determining whether they store the reference value of the same enum constant—that is, whether the references are aliases. Thus for any two enum references, `meal1` and `meal2`, the expressions `meal1.equals(meal2)` and `meal1 == meal2` are equivalent.

The `java.lang.Enum` class also overrides the `toString()` method from the `Object` class. The `toString()` method returns the name of the enum constant, but it is *not* `final` and can be overridden by an enum type—but that is rarely done. Examples in this subsection illustrate the use of these methods.

Extending Enum Types: Constant-Specific Class Bodies

Constant-specific class bodies define anonymous classes (§9.6, p. 521) inside an enum type—they implicitly extend the enclosing enum type creating new subtypes. The enum type `Meal` in Example 5.26 declares constant-specific class bodies for its constants. The following skeletal code declares the constant-specific class body for the enum constant `BREAKFAST`:

Click here to view code image

```
  BREAKFAST(7,30) {                    // (1) Start of constant-specific class 
    @Override
    public double mealPrice(Day day) {  // (2) Overriding abstract method
      ...
    }
    @Override
    public String toString() {          // (3) Overriding method from the Enum c
      ...
    }
  }                                     // (4) End of constant-specific class bo
```

The constant-specific class body, as the name implies, is a class body that is specific to a particular enum constant. As for any class body, it is enclosed in curly brackets, `{ }`. It is declared immediately after the enum constant and any constructor arguments. In the code above, it starts at (1) and ends at (4). Like any class body, it can contain member declarations. In the above case, the body contains two method declarations: an implementation of the method `mealPrice()` at (2) that overrides the `abstract` method declaration at (7) in the enclosing enum supertype `Meal`, and an implementation of the `toString()` method at (3) that overrides the one inherited by the `Meal` enum type from the superclass `java.lang.Enum`. The `@Override` annotation used on these overriding methods ensures that the compiler will issue an error message if such a method declaration does not satisfy the criteria for overriding.

The constant-specific class body is an anonymous class—that is, a class with no name. Each constant-specific class body defines a distinct, albeit anonymous, subtype of the enclosing enum type. In the code above, the constant-specific class body defines a subtype of the `Meal` enum type. It inherits members of the enclosing enum supertype that are not private, overridden, or hidden. When the enum type `Meal` is loaded at runtime, this constant-specific class body is instantiated, and the reference value of the instance is assigned to the enum constant `BREAKFAST`. Note that the type of the enum constant is `Meal`, which is the supertype of the anonymous subtype represented by the constant-specific class body. Since supertype references can refer to subtype objects, this assignment is legal.

Each enum constant overrides the `abstract` method `mealPrice()` declared in the enclosing enum supertype—that is, provides an implementation for the method. The compiler will report an error if this is not the case. Although the enum type declaration specifies an `abstract` method, the enum type declaration is *not* declared `abstract`—contrary to an `abstract` class. Given that the references `meal` and `day` are of the enum types `Meal` and `Day` from Example 5.26, respectively, the method call

```
meal.mealPrice(day)
```

will execute the `mealPrice()` method from the constant-specific body of the enum constant denoted by the reference `meal`.

Two constant-specific class bodies, associated with the enum constants `BREAKFAST` and `LUNCH`, override the `toString()` method from the `java.lang.Enum` class. Note that the `toString()` method is not overridden in the `Meal` enum type, but in the anonymous classes represented by two constant-specific class bodies. The third enum constant, `DINNER`, relies on the `toString()` method inherited from the `java.lang.Enum` class.

Constructors, `abstract` methods, and `static` methods cannot be declared in a constant-specific class body. Instance methods declared in constant-specific class bodies are only accessible if they override methods in the enclosing enum supertype.

Example 5.26 *Declaring Constant-Specific Class Bodies*

Click here to view code image

```
// File: Day.java
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Click here to view code image

```
// File: Meal.java
public enum Meal {
  // Each enum constant defines a constant-specific class body
  BREAKFAST(7,30) {                                                     // (1)
    @Override
    public double mealPrice(Day day) {                                  // (2)
      double breakfastPrice = 10.50;
      if (day.equals(Day.SATURDAY) || day == Day.SUNDAY)
        breakfastPrice *= 1.5;
      return breakfastPrice;
    }
    @Override
    public String toString() {                                          // (3)
      return "Breakfast";
    }
  },                                                                    // (4)
  LUNCH(12,15) {
    @Override
    public double mealPrice(Day day) {                                  // (5)
      double lunchPrice = 20.50;
      switch (day) {
        case SATURDAY: case SUNDAY:
          lunchPrice *= 2.0;
      }
      return lunchPrice;
    }
    @Override
    public String toString() {
      return "Lunch";
    }
  },
  DINNER(19,45) {
    @Override
    public double mealPrice(Day day) {                                // (6)
      double dinnerPrice = 25.50;
      if (day.compareTo(Day.SATURDAY) >= 0 && day.compareTo(Day.SUNDAY) <= 0)
        dinnerPrice *= 2.5;
      return dinnerPrice;
    }
  };

  // Abstract method implemented in constant-specific class bodies.
  abstract double mealPrice(Day day);                                 // (7)

  // Enum constructor:
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Instance fields: Time for the meal.
  private int hh;
  private int mm;

  // Instance methods:
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                              // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}

// File: MealPrices.java
public class MealPrices {

  public static void main(String[] args) {                                // (8
    System.out.printf(
        "Please note that %s, %s, on %s costs $%.2f.%n",
        Meal.BREAKFAST.name(),                                             // (9
        Meal.BREAKFAST.getTimeString(),
        Day.MONDAY,
        Meal.BREAKFAST.mealPrice(Day.MONDAY)                               // (1
    );

    System.out.println("Meal prices on " + Day.SATURDAY + " are as follows:");
    Meal[] meals = Meal.values();
    for (Meal meal : meals) {
      System.out.printf(
          "%s costs $%.2f.%n", meal, meal.mealPrice(Day.SATURDAY)        // (1
      );
    }
  }
}
```

Output from the program:

Click here to view code image

```
Please note that BREAKFAST, 07:30, on MONDAY costs $10.50.
Meal prices on SATURDAY are as follows:
Breakfast costs $15.75.
Lunch costs $41.00.
DINNER costs $63.75.
```

In Example 5.26, the `mealPrice()` method declaration at (2) uses both the `equals()` method and the `==` operator to compare enum constants for equality. The `mealPrice()` method declaration at (5) uses enum constants in a `switch` statement. Note that the `case` labels in the `switch` statement are enum constant names, without the enum type name. The `mealPrice()` method declaration at (6) uses the `compareTo()` method to compare enum constants.

The `main()` method at (8) in Example 5.26 demonstrates calling the `mealPrice()` method in the constant-specific class bodies. The `mealPrice()` method is called at (10) and (11). Example 5.26 also illustrates the difference between the `name()` and the `toString()` methods of the enum types. The `name()` method is called at (9), and the `toString()` method is called implicitly at (11) on `Meal` enum values. The `name()` method always prints the enum constant name exactly as it was declared. Which `toString()` method is executed depends on whether the `toString()` method from the `java.lang.Enum` class is overridden. Only the constant-specific class bodies of the enum constants `BREAKFAST` and `LUNCH` override this method. The output from the program confirms this to be the case.

Enum Values in Exhaustive `switch` Expressions

A `switch` expression is always *exhaustive*—that is, the cases defined in the `switch` expression must cover *all* values of the selector expression type or the code will not compile (§4.2, p. 160).

The `switch` expression below at (1) is exhaustive, as all values of the enum type `Day` are covered by the two switch rules at (2) and (3). Deleting any one or both of these switch rules will result in a compile-time error.

Click here to view code image

```
Day day = Day.MONDAY;             // See Example 5.26, p. 295, for enum type D
String typeOfDay = switch (day) {                                           //
  case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";           //
  case SATURDAY, SUNDAY                             -> "Weekend";           //
};
```

The `switch` expression below at (4) is also exhaustive, as all values of the enum type `Day` are collectively covered by the `case` label at (5) and the `default` label at (6). Deleting the `case` label at (5) does *not* result in a compile-time error, as the `default` label will then cover all values of the selector expression—but it may result in a wrong value being returned by the `switch` expression. However, deleting the `default` label at (6) will result in a compile-time error, as the `case` label at (5) only covers a subset of the values for the `Day` enum type. Compile-time checking of exhaustiveness of a `switch` expression whose switch rules are defined by the arrow notation results in robust and secure code.

Click here to view code image

```
typeOfDay = switch (day) {                                                  //
  case SATURDAY, SUNDAY -> "Weekend";                                       //
  default -> "Weekday";                                                     //
}
```

Declaring Type-Safe Enums, Revisited

We have seen declarations of enum types as top-level types, but they can also be nested as `static` member and `static` local types (§9.1, p. 491). Although nested enum types are implicitly `static`, they can be declared with the keyword `static`. The following skeletal code shows the two enum types `Day` and `Meal` declared as `static` member types in the class `MealPrices`:

Click here to view code image

```
public class MealPrices {
  public enum Day { /* ... */ }                            // Static member

  public static enum Meal { /* ... */ }                    // Static member

  public static void main(String[] args) { /* ... */ }   // Static method
}
```

An enum type cannot be explicitly extended using the `extends` clause. An enum type is implicitly `final`, unless it contains constant-specific class bodies. If it declares constant-specific class bodies, it is implicitly extended. No matter what, it cannot be explicitly declared `final`.

An enum class is either *implicitly* `final` if its declaration contains no enum constants that have a class body, or *implicitly* `sealed` if its declaration contains at least one enum constant that has a class body. Since an enum type can be either implicitly `final` or implicitly `sealed`, it can implement a sealed interface—in which case, it must be specified in the `permits` clause of the sealed interface (p. 311).

An enum type *cannot* be declared `abstract`, regardless of whether each `abstract` method is overridden in the constant-specific class body of every enum constant.

Like a class, an enum can implement interfaces.

Click here to view code image

```
public interface ITimeInfo {
  public int getHour();
  public int getMins();
}

public enum Meal implements ITimeInfo {
  // ...
  @Override public int getHour() { return this.hh; }
  @Override public int getMins() { return this.mm; }
  // ...
}
```

The Java Collections Framework provides a special-purpose *set* implementation (`java.util.EnumSet`) and a special-purpose *map* implementation (`java.util.EnumMap`) for use with enum types. These implementations provide better performance for enum types than their general-purpose counterparts, and are worth checking out.

5.14 Record Classes

Exchanging *plain data* between programs is a common task. *Plain data objects* allow efficient *aggregation* of data and provide transparent *access* to it. In the literature, such data objects are referred to by different names; the acronym *POJOs* (*Plain Old Java Objects*) is common in the Java community.

A *plain data class* can always be implemented as a normal class, but this is tedious and repetitive, entailing boilerplate code and full weight of the OOP programming model. In Example 5.27, the class `CD_v1` represents the data for a CD. A full-fledged implementation of this data class requires declaration of its instance fields that constitute the data, an appropriate constructor to initialize the data fields, *get methods* to allow access to the data, and overriding the pertinent methods from the `Object` class to make `CD` objects more meaningful and useful. (For details on overriding methods from the

`Object` class, see Chapter 14, p. 741). Although most IDEs can generate the code necessary to implement such data objects, they lack the incremental validation required when changes are made to the data model of the normal class.

Example 5.27 *The* `CD` *Class*

Click here to view code image

```
package record.basics;
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
```

Click here to view code image

```
package record.basics;
import java.time.Year;
import java.util.Objects;

/** A class that represents a CD. */
public class CD_v1 {

  // Instance fields:
  private final String artist;      // Name of the artist.
  private final String title;       // Title of the CD.
  private final int    noOfTracks;  // Number of tracks on the CD.
  private final Year   year;        // Year the CD was released.
  private final Genre  genre;       // Music genre of the CD.

  // Non-zero argument constructor:
  public CD_v1(String artist, String title, int noOfTracks,
               Year year, Genre genre) {
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
    this.genre      = genre;
  }

  // Get methods:
  public String getArtist()     { return this.artist; }
  public String getTitle()      { return this.title; }
  public int    getNoOfTracks() { return this.noOfTracks; }
  public Year   getYear()       { return this.year; }
  public Genre  getGenre()      { return this.genre; }

  // Overridden methods from the Object class:
  @Override public String toString() {
    return String.format("<%s, \"%s\", %d, %s, %s>",
        this.artist, this.title, this.noOfTracks, this.year, this.genre);
  }

  @Override public boolean equals(Object obj) {
    return (this == obj)
        || (obj instanceof CD_v1 other
            && this.artist.equals(other.artist)
            && this.title.equals(other.title)
            && this.noOfTracks == other.noOfTracks
            && this.year == other.year
            && this.genre == other.genre);
  }

  @Override public int hashCode() {
    return Objects.hash(this.artist, this.title, this.noOfTracks,
                        this.year, this.genre);
  }
}
```

Record Class Basics

A *record class* in Java is a special-purpose class that simplifies declaration and handling of *an aggregate of values* that comprise the state of a plain data object. A record class defines immutable fields and the compiler generates the *get methods* (also called *getter* or *accessor methods*) necessary to access the values of fields in instances of the record class. A record class generally does not provide any heavy processing of the data; the primarily goal is to allow users to access the data.

The basic syntax for a record class is shown below. The *record header* of a top-level record class can specify the `public` access modifier, as in a top-level normal class. The contextual keyword `record` is used in the header to declare a record class. It has this special meaning only in this context. A record class has a *name*, similar to a normal class. The header specifies a comma-separated *component list* that comprises *field declarations* (called *record components*), where each field declaration specifies the *name* and the *type* of the field. For the basic record class declaration, the record body can be left empty.

Click here to view code image

```
access_modifier record record_name(component_list)        // Record header
{ }                                                         // Empty record body
```

The `CD_v0` class in Example 5.27 can be declared as a record class, as shown in Example 5.28. The `public` record class `CD` declared at (1) in Example 5.28 is equivalent to the normal class `CD_v1` in Example 5.27. The compiler automatically generates the necessary fields, contructor, and method declarations for the record class `CD`. The client class `DataUser` in Example 5.28 utilizes data objects of record class `CD`.

Example 5.28 *The* `CD` *Record Class*

Click here to view code image

```
package record.basics;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,          // (1)
                 Year year, Genre genre) { /* Empty body */ }
```

Click here to view code image

```
package record.basics;
import java.time.Year;

public class DataUser {
  public static void main(String[] args) {
    // Some ready-made CDs:                                                (2)
    CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.PO
    CD cd1 = new CD("Jaav",      "Java Jam",        6, Year.of(2017), Genre.JA
    CD cd2 = new CD("Funkies",   "Lambda Dancing", 10, Year.of(2018), Genre.PO
    CD cd3 = new CD("Genericos", "Keep on Erasing", 8, Year.of(2018), Genre.JA
    CD cd4 = new CD("Genericos", "Hot Generics",   10, Year.of(2018), Genre.JA

    // An array of CDs.
    CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};

    System.out.println("     Artist    Title           No. Year Genre");
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
          cd.artist(), cd.title(), cd.noOfTracks(),                     // (3)
          cd.year(), cd.genre());
      System.out.printf("cd%d: %s%n", i, cdToString);
    }

    System.out.println();
    System.out.println(cd0.toString());                                 // (4)

    CD cdX = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.PO
    System.out.println("cd0.equals(cdX): " + cd0.equals(cdX));          // (5)
  }
}
```

Output from the program:

Click here to view code image

```
      Artist    Title           No. Year Genre
cd0: Jaav      Java Jive       8   2017 POP
cd1: Jaav      Java Jam        6   2017 JAZZ
cd2: Funkies   Lambda Dancing  10  2018 POP
cd3: Genericos Keep on Erasing 8   2018 JAZZ
cd4: Genericos Hot Generics    10  2018 JAZZ

cd0: CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
cd0.equals(cdX): true
```

The compiler automatically generates the necessary declarations for a record class when it is declared according to the simplified syntax form, in particular, when the record body is empty.

- Each record component in the component list results in a `private`, `final` *instance field* in the record. In other words, these *component fields* pertain to objects of the record class, and their values constitute the state of a record. From the `CD` record class declaration, the following component fields are created: `String artist`, `String title`, `int noOfTracks`, `Year year`, and `Genre genre`. The component fields in the `CD` record class are equivalent to the instance field declarations in the `CD_v0` class in Example 5.27.

Fields being `final` means that once a `final` field is initialized, its value cannot be changed. However, if a field refers to a mutable object, this object can be modified, but the reference value of the object assigned to the field cannot be changed. Record classes are thus said to be *shallowly immutable*.

The component fields in the `CD` record class have immutable values, as objects of `String`, `Year`, and `Genre` are immutable, as are primitive values. The `CD` record class is therefore immutable. (See also §6.7, p. 356, for a discussion on immutability.)

As we shall see, the *order* of the record components in the component list has implications in the record class.

- Each record component results in a `public` zero-argument *get method* (also known as a *getter* method). It is important to note that the get method has the same name and return type as the corresponding record component. This does not create any problem, as method names and field names are in different *name spaces*.

  The compiler thus ensures that each record component has a corresponding `private`, `final` component field and a read-only `public` zero-argument get method that returns the value of this field.

  The following `public` zero-argument get methods are created for the `CD` record class: `String artist()`, `String title()`, `int noOfTracks()`, `Year year()`, and `Genre genre()`, each having the name and the return type as declared by the corresponding record component. As expected, calling a get method returns the value of the corresponding component field. In Example 5.28, the get methods are called at (3) to obtain the values of individual component fields in a `CD` record. Although their names are different, the get methods of the `CD` record class are equivalent to the get methods of the `CD_v0` class in Example 5.27.

- In order to initialize all component fields properly, the compiler automatically creates a `public` constructor—called the *implicit canonical constructor*—that has a parameter list that corresponds to the record components in *name, type* and *order*. A call to the canonical constructor in a `new` expression results in the argument values being assigned to the corresponding component fields in the same order. The implicit canonical constructor of a record class functions analogously to the default constructor in a normal class.

The implementation of the implicit canonical constructor for the `CD` record class is equivalent to the non-zero argument normal constructor implemented in the `CD_v0` class in Example 5.27.

Creating records is no different from creating objects of a normal class. In Example 5.28, `CD` records are constructed by calling the implicit canonical constructor at (2) in the `new` expression. Note that the order of the specified values corresponds to that of the record components in the component list.

Click here to view code image

```
CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre
...
```

- The compiler also creates an implementation of the `toString()` method that is overridden from the `Object` class. The `toString()` method creates a default text representation of a record that includes the name of the record class and a textual list with name-value pairs for each component field in the order specified in the component list.

  In Example 5.28, the `toString()` method is explicitly called in the print statement at (4), resulting in the following output:

  Click here to view code image

  ```
  CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
  ```

- An implementation of the `equals()` method (§14.2, p. 744) that is overridden from the `Object` class is also created to compare two `CD` records for equality, based on the values of corresponding component fields in the two records. The method compares the component fields in the same order as specified in the component list, and returns `true` if the values in corresponding fields all match—analogous to the implementation of the `equals()` method in the `CD_v0` class in Example 5.27. Overriding of the `equals()` method is essential if records are to be used in collections (§15.12, p. 866).

In Example 5.28, two records `cd0` and `cdX` are compared for equality with the `equals()` method, showing that the two records are equal, since they have the same state.

- Lastly, the compiler generates an implementation of the `hashCode()` method (§14.3, p. 753) that is overridden from the `Object` class. The method returns a hash code that is computed based on the values of the component fields, in the same order as specified in the component list—analogous to the implementation of the `hashCode()` method in the `CD_v0` class in Example 5.27. Overriding of the `hashCode()` method is essential if records are to be used in sets and maps (§15.8, p. 830).

Restrictions on Record Declarations

A record class is implemented as an implicitly `final` direct subclass of the `java.lang.Record` abstract class that defines the default inherited behavior of such classes. In particular, the `Record` class defines `abstract` methods that override the `equals()`, `hashCode()`, and `toString()` methods of the `Object` class. This means that every record class must implement these methods either explicitly or implicitly. We have seen in Example 5.28 that the compiler provides the implementation of these methods automatically when any of these methods is not declared explicitly in the record class.

A record class cannot have an explicit `extends` clause to declare a direct superclass, not even its direct superclass `Record`. Being `final` also means that a record class cannot be declared `abstract`. However, a record class is seldom declared explicitly `final`. As such, a record class is solely defined by its state and cannot be extended by any subclass.

Note also that the name of a record component in the component list cannot be the same as the name of any of the methods in the `Object` class, except `equals`.

Augmenting Basic Record Class Declaration

In the rest of this section, we explore how the basic declaration of a record class can be augmented and customized. The general syntax of a record class is shown below.

Click here to view code image

```
access_modifier record record_name(component_list) optional_implements_clause {
  optional_constructor_declarations
  optional_member_declarations
}
```

The basic declaration of a record class can be augmented with an `implements` clause in order to implement any interfaces. Constructors can be customized, as can the get methods corresponding to the component fields. New instance methods can be declared, as can static members and initializers, but new instance fields and initializers are not allowed in the record body.

Record Constructors

In this section we discuss three kinds of constructors that can be specified in a record class declaration:

- A *normal canonical record constructor*, if declared, supersedes the implicit canonical record constructor.
- If a normal canonical record constructor is not specified, a *compact canonical record constructor* can be declared. In this case, the implicit canonical record constructor is generated. Note that either the normal canonical record constructor or the compact canonical record constructor can be specified, but not both.
- Any number of *non-canonical record constructors* can be specified, whether or not any canonical constructor is provided.

Normal Canonical Record Constructor

In a basic record class declaration, the compiler generates the *implicit canonical record constructor* discussed earlier to ensure that component fields of a record are initialized properly. This canonical record constructor can be implemented *explicitly* when there is a need to process the argument values before they are assigned to the component fields. This explicit canonical record constructor is called the *normal canonical record constructor* as its declaration resembles the constructor of a normal class. If a record class implements the normal canonical record constructor, the implicit counterpart is *not* generated.

The normal canonical constructor must have the *same parameter list* as the implicit canonical record constructor, in *name*, *type*, and *order*. In the constructor, the component fields must be accessed with the `this` reference to distinguish them from the parameter names, and *all* component fields must be initialized in the constructor. In Example 5.29, the record class `CD` provides an implementation of the normal canonical constructor at (2). The parameter values are sanitized at (3) before being assigned to the component fields at (4). Running the `DataUser` class in Example 5.28 with the record class `CD` in Example 5.29 produces the same results.

Here are a few other restrictions to note regarding the normal canonical constructor:

- It must be at least as accessible as the record class. For example, for a top-level record class that has `public` accessibility, it must be declared `public`.
- It cannot be generic (§11.7, p. 593), or have a `throws` clause in its header (§7.5, p. 388). In other words, any checked exception thrown by the constructor must be handled in the body of the constructor.
- It cannot invoke another record constructor with the `this()` expression to chain constructors locally (p. 209), as is the case with normal constructors.

Example 5.29 *Normal Canonical Record Constructor*

Click here to view code image

```
package record.constructors.canonical;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,          // (1)
                 Year year, Genre genre) {

  // Normal canonical record constructor                               // (2)
  public CD(String artist, String title, int noOfTracks, Year year, Genre genr
    // Sanitize the parameter values:                                     (3)
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;
    // Initialize all component fields:                                   (4)
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
```

```
        this.genre       = genre;
      }
    }
```

Compact Canonical Record Constructor

The *compact canonical record constructor* is a more concise form of the normal canonical record constructor. It eliminates the need for a parameter list and the initialization of the component fields. The parameter list is derived from the component list declared in the record header, and the initialization of *all* component fields is left to the *implicit* canonical record constructor. In fact, the component fields cannot be accessed in the compact constructor, as the component names refer to the parameter names of the implicit canonical constructor. Note that the implicit canonical record constructor is generated when the compact constructor is provided in the record class declaration. The compact constructor is called in a `new` expression, exactly as any other constructor, passing the argument values in the call.

In Example 5.30, the compact canonical constructor for the record class `CD` is implemented at (2). It has no parameter list and it does *not* initialize the component fields. It only sanitizes the values passed to the constructor. Again running the `DataUser` class in Example 5.28 with the record class `CD` in Example 5.30 produces the same results.

The *implicit* canonical record constructor is called before exiting from the compact canonical constructor; therefore, a `return` statement is not allowed in the compact constructor. The compact constructor primarily functions as a validator for the data values before they are assigned to the component fields by the *implicit* canonical record constructor.

Previous restrictions mentioned for the normal canonical record constructor also apply to the compact canonical record constructor.

Example 5.30 *Compact Record Constructor and Non-Canonical Constructor*

Click here to view code image

```
package record.constructors.compact;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,           // (
                 Year year, Genre genre) {
```

```
  // Compact canonical record constructor                                    (
  public CD {
    // Sanitize the values passed to the constructor:                        (
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;

    // Cannot explicitly assign to component fields:                         (
    // this.artist = artist;                    // Compile-time error!
  }

  // A non-canonical record constructor                                      (
  public CD() {
    this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (
  }
}
```

Normal Non-Canonical Record Constructors

A record class declaration can specify any number of *non-canonical record constructors*—that is, constructors whose signature is *not* the same as that of the canonical constructor.

In Example 5.30, a non-canonical constructor for the record class `CD` is implemented at (5). It is a zero-argument non-canonical record constructor to create a record with default values for the component fields.

The first statement in the non-canonical constructor at (6) is an explicit invocation of the canonical constructor with the `this()` expression. The signature of the `this()` expression matches that of the canonical constructor in type and in order. The rule is that chaining of constructors with the `this()` expression (p. 209) must ultimately lead to the canonical constructor being invoked so that all component fields are initialized. In Example 5.30, the constructor invocation at (6) leads to the compact constructor being invoked, and whose execution in turn leads to the implicit canonical record constructor being invoked. Use of this non-canonical constructor is illustrated in the following print statement:

Click here to view code image

```
  System.out.println(new CD());        // Calls the non-canonical constructor
```

Output from the print statement shows that the record was created after the string values were sanitized:

Click here to view code image

```
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
```

Non-canonical constructors are primarily used for creating specialized records.

The restrictions noted for the normal and the compact canonical constructors do *not* apply to a non-canonical constructor: It can be less accessible than its record class, be generic, and can specify a `throws` clause, but it must invoke another record constructor with the `this()` expression.

Member Declarations

The component fields of a record class are always automatically generated based on the field components specified in the component list. Thus a record class *cannot* declare any new instance fields in addition to those specified in the component list.

However, new instance methods can be declared in a record class. In Example 5.31, three new instance methods are declared at (8) in the record class to determine the genre of a CD.

Automatic generation of any get method or methods overridden from the `Object` class (`equals()`, `hashCode()`, and `toString()`) will be suppressed in a record class, if the record class provides an implementation for any of these methods.

In Example 5.31, the `CD` record class provides an implementation for the `title()` method at (9) that returns the uppercase version of the string in the `title` field. This explicit method will be invoked when called by a client to get the value of the `title` field in a `CD` record. An explicit get method can be annotated with the `@Override` annotation to enable validation of its method signature against that of the default get method that would otherwise be generated. Such an explicit get method must fulfill the following criteria:

- It must be an instance method that is declared `public`.
- It must have no formal parameters or a `throws` clause.
- Its return type must be the same as the declared type of its corresponding record component.
- It must not be a generic method.

When the object referenced by a field is mutable, its get method can be explicitly implemented to return a copy of the object so that the original object is never modified, maintaining the immutability of the record. As values of individual component fields cannot be changed, a new record with any modified values must be created.

Also, `static` fields, methods, and initializers (§10.7, p. 545) can be declared. In Example 5.31, the `CD` record class declares `static` fields to create some ready-made `CD` records. Argument values passed in the constructor call to create `CD` records will be sanitized by the compact constructor at (2) before being assigned to the component fields by the implicit canonical record constructor.

Nested static types can also be declared in a record class (Chapter 9, p. 489). In Example 5.31, the enum type `Genre` is declared as a `static` member of the `CD` record class at (11), and is used in the `CD` record class.

In Example 5.31, the client class `DataUser2` uses some of the explicitly implemented methods in the `CD` record class. The `CD.isOther()` method is used at (13) to filter an array of CDs created at (12). The output from the program shows that the values passed in the constructor calls were sanitized.

An uppercase version of the string in the `title` field is printed when the `title()` method is explicitly invoked on a `CD` record. However, note that the automatically generated `toString()` method, called at (14), does not use the uppercase version of the title. The reason is that the fields are accessed directly by the generated methods in the record class, and not by calling the get methods. The `toString()` method accesses the `title` field directly, thereby accessing the original string in the field. In order to use the uppercase version of the string in all contexts, it can be converted to uppercase in the compact constructor by replacing line (4a) with (4b) so that the uppercase version of the string is assigned to the `title` field—and thereby making the `title()` method declared at (9) redundant.

Example 5.31 *Other Member Declarations in a Record Class*

Click here to view code image

```
package record.customize;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,              // (
                 Year year, Genre genre) {

  // Compact canonical record constructor                                     (
  public CD {
    // Sanitize the values passed to the constructor:                         (
    artist = artist.strip();
    title = title.strip();                                                // (
//  title = title.strip().toUpperCase();                                  // (
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;

// Cannot explicitly assign to component fields:                             (
// this.artist = artist;                                  // Compile-time erro
  }

  // A non-canonical record constructor                                      (
  public CD() {
    this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (
  }

  //  New instance methods:                                                  (
  public boolean isPop()   { return this.genre == Genre.POP; }
  public boolean isJazz()  { return this.genre == Genre.JAZZ; }
  public boolean isOther() { return this.genre == Genre.OTHER; }

  // Customize a get method:                                                 (
  @Override public String title() {
     return this.title.toUpperCase();
  }

  // Static fields with some ready-made CDs:                                 (
  public final static CD cd0
      = new CD("   Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP)
  public final static CD cd2
      = new CD("Funkies ",   " Lambda Dancing  ", 10, Year.of(2024), null);
  public final static CD cd4
      = new CD("Genericos", "Hot Generics",   -5, Year.of(2018), Genre.JAZZ);

  // Declare a nested type:
  public enum Genre {POP, JAZZ, OTHER}                                   //   (
}
```

Click here to view code image

```
package record.customize;

public class DataUser2 {
  public static void main(String[] args) {
    CD[] cdArray = {CD.cd0, CD.cd2, CD.cd4, new CD()};                   // (1
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      if (cd.isOther()) {                                                // (1
        System.out.println(cd.toString());                               // (1
      }
    }
    System.out.println("Title: " + cdArray[1].title());                  // (1
  }
}
```

Output from the program:

Click here to view code image

```
CD[artist=Funkies, title=Lambda Dancing, noOfTracks=10, year=2022, genre=OTHER
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
Title: LAMBDA DANCING
```

Other Aspects of Record Classes

Some other aspects of record classes are mentioned below, and are covered in detail elsewhere in the book.

Implementing Interfaces

Records can implement interfaces, which is no different from a normal class implementing interfaces. As a record class is implicitly `final`, it cannot extend other classes. The `CD` record class below implements the `Serializable` interface and the `Comparable<CD>` interface. Example 20.6, p. 1263, uses the `CD` record class to demonstrate record serialization.

Click here to view code image

```
public record CD(String artist, String title, int noOfTracks,          // (1)
    Year year, Genre genre) implements Serializable and Comparable<CD> {

  @Override public int compareTo(CD other) { /* See Example 16.1, p. 883. */ }

  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

The `CD` record class in Example 16.1, p. 883, implements the `Comparable<CD>` interface so that `CD` records can be compared.

Record Serialization

Example 20.6, p. 1263, demonstrates serializing records to external storage. It is important to note that serialization of records cannot be customized, as with objects of normal classes, and that a record is always deserialized using the canonical constructor.

Generic Record Classes

Generic record classes can be delcared, analogous to generic classes (Chapter 11, p. 563). The generic record class `Container` below has one type parameter (`T`):

Click here to view code image

```
record Container<T>(T item) { /* Empty body */ }
```

We can create records by parameterizing the generic record class:

Click here to view code image

```
Container<String> p0 = new Container<>("Hi");
Container<Integer> p1 = new Container<>(Integer.valueOf(10));
```

Nested Record Classes

Record classes can be declared as nested record classes—in particular, as `static` *member types* and as `static` *local types*. For details on nested record classes, see Chapter 9, p. 489.

Direct Permitted Subtypes of Sealed Interfaces

A record class is only allowed in the `permits` clause of a *sealed interface*, if it implements the sealed interface. As a record class is implicitly `final`, it also fulfills the second criterion for a permitted direct subtype (p. 311).

Finally, it is worth noting that records in Java are designed so that the record state solely defines the representation of the record. Record classes require less effort to implement than normal classes and have an efficient memory footprint and runtime performance. Also, their immutability provides thread safety in concurrent applications.

5.15 Sealed Classes and Interfaces

Design by inheritance promotes code reuse in the OOP model—where subtypes can inherit code from their supertypes. Such an inheritance hierarchy is depicted in Figure 5.8, where the subclasses `PrintedBook`, `Ebook`, and `Audiobook` can inherit and reuse code from the superclass `Book`. The inheritance hierarchy of the `Book` superclass can also be regarded as a *domain model* for books (albeit greatly simplified for this exposition). However, this domain model as represented by the inheritance hierarchy in Figure 5.8 does not give the `Book` superclass much control over which classes can be classified as books. It can easily be exploited by clients to create their own subclasses of the `Book` superclass, thereby undermining the domain model represented by the inheritance hierarchy in Figure 5.8. If the `PrintedBook` subclass is not declared `final`, it can also be freely extended to define more kinds of books.

The superclass `Book` in Figure 5.8 cannot control which subclasses belong to the book domain, as it cannot control its extensibility. When it comes to domain modeling using inheritance, modifiers such as `abstract` and `final` are not adequate, nor are the accessibility modifiers to control the use of the superclass. In this section, we look at how sealed classes and interfaces aid in domain modeling that promotes accessibility of the superclass and allows control of extensibility of its subclasses. Sealed classes also have the additional benefits of resulting in secure hierarchies and aiding the compiler to provide better analysis of potential problems in the code.

Figure 5.8 *A Domain Model as Represented by Inheritance Hierarchy*

Sealed Classes

Sealing of classes and interfaces allows fine-grained control over the inheritance hierarchy of reference types in a given domain. A sealed class or interface only allows specific classes or interfaces to extend or implement its definition. To implement sealed classes, the language has introduced two new modifiers, `sealed` and `non-sealed`, and the `permits` clause—these three identifiers are *contextual keywords* that have special meaning only in the context of defining sealed types. As we shall see, the modifier `final` is also an integral part of defining sealed types.

We first look at sealed classes, and later we discuss sealed interfaces and compare the two (p. 315).

Going back to Figure 5.8 with the simplified domain model of books, we want to ensure that only the subclasses `PrintedBook`, `Ebook`, and `Audiobook` can *directly extend* the abstract superclass `Book` (i.e., ensure that the three subclasses are the only *permitted direct subclasses* of the `sealed` class `Book`). Figure 5.9 shows the modeling of the inheritance hierarchy with the required constraints. In this example, the classes involved are located in the same package. Skeletal code for the classes in shown in the discussion below, and complete class declarations can be found in Example 5.32.

Figure 5.9 *Sealed Classes for Domain Modeling of Inheritance Hierarchy*

A sealed class is declared using the modifier `sealed` and the optional `permits` clause in the class header. The `permits` clause is optional if a sealed class and its permitted direct subclasses are declared in the same compilation unit (p. 317). If a class specifies the `permits` clause, the class must be declared `sealed`. The `permits` clause, if specified, must appear after any `extends` and `implements` clauses in the class header.

The superclass `Book` is declared `sealed` and explicitly specifies a `permits` clause with a list of its permitted direct subclasses, as shown at (1) below. Keep in mind that since these subclasses are `public`, they are declared in separate compilation units, but belong to the same package.

Click here to view code image

```
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {/*(1)
```

Each *permitted direct subclass* of a sealed class is bound by the following *contract*:

- Each permitted direct subclass *must extend* its direct `sealed` superclass. Note that the sealing relationship is defined between two consecutive levels in the inheritance hierarchy: between a superclass and its direct subclasses. This is also the case for the permitted direct subclasses of the `Book` superclass:

  Click here to view code image

  ```
  public non-sealed class PrintedBook extends Book {/*...*/}

  public final class Ebook extends Book {/*...*/}

  public final class Audiobook extends Book {/*...*/}
  ```

- Each permitted direct subclass must be declared either `sealed`, `non-sealed`, or `final`. This avoids any unwanted subclassing to the inheritance hierarchy of a sealed class.

  A permitted direct subclass that is declared `final` cannot be extended, as in the case of a `final` normal class. This implies that the inheritance hierarchy of a `final` permitted subclass cannot be extended. The permitted direct subclasses `Ebook` and `Audiobook` are declared `final`. Any attempt to extend them would result in a compile-time error.

  Click here to view code image

  ```
  class EComicBook extends Ebook {}   // Compile-time error!
  ```

  A permitted direct subclass that is declared `non-sealed` can be *freely extended*. This implies that the inheritance hierarchy of a `non-sealed` permitted subclass can be extended. Note that a class cannot be declared `non-sealed` unless it is a permitted direct subclass of a direct `sealed` superclass.

  The permitted direct subclass `PrintedBook` is declared `non-sealed`. Client code can readily extend this class:

Click here to view code image

```
class Hardcover extends PrintedBook {}
class Paperback extends PrintedBook {}
```

A `sealed` permitted direct subclass applies sealing to its own permitted direct subclasses, just like its direct `sealed` superclass. This implies that the inheritance hierarchy of a `sealed` permitted subclass can be extended, but in a restricted way. For example, if the permitted subclass `PrintedBook` was declared `sealed`, its permitted direct subclasses would be subject to the same rules for declaring permitted subclasses, as shown below:

Click here to view code image

```
public sealed class PrintedBook extends Book permits Hardcover, Paperb
final class Hardcover extends PrintedBook {}
final class Paperback extends PrintedBook {}
```

- Each permitted direct subclass must be *accessible* by the direct `sealed` superclass.

  If the `sealed` superclass is in a *named module*, the permitted subclasses must also be in the same module (Chapter 19, p. 1161). Note that this does not mean that they have to be in the same package in the named module. Their *fully qualified names* can be used in the `permits` clause to access the subclasses in their respective packages (§6.3, p. 326).

  If the `sealed` superclass is in *the unnamed module*, the permitted subclasses must all be in either the *same named package* or *the unnamed package* as the `sealed` superclass. In Example 5.32, the `sealed` superclass `Book` is in a named package (but in the unnamed module), as are the permitted subclasses, specified using the `package` statement (§6.3, p. 326).

Example 5.32 *Declaring* `sealed`, `non-sealed`, *and* `final` *Classes*

Click here to view code image

```
package signedandsealed;

public abstract sealed class Book permits PrintedBook, Ebook, Audiobook  { //
  private String isbn;
  protected Book(String isbn) {
    this.isbn = isbn;
  }
  public String getIsbn() { return this.isbn; }
}
```

Click here to view code image

```
package signedandsealed;

public non-sealed class PrintedBook extends Book {                           //
  private int pageCount;
  protected PrintedBook(String isbn, int pageCount) {
    super(isbn);
    this.pageCount = pageCount;
  }
  public int getPageCount() { return this.pageCount; }
}
```

Click here to view code image

```
package signedandsealed;

public final class Ebook extends Book {                                      //
  private String format;
  public Ebook(String isbn, String format) {
    super(isbn);
    this.format = format;
  }
  public String getFormat() { return this.format; }
}
```

Click here to view code image

```
package signedandsealed;

public final class Audiobook extends Book {                                  //
  private String narrator;
  private double length;
  public Audiobook(String isbn, String narrator, double length) {
    super(isbn);
    this.narrator = narrator;
    this.length = length;
  }
  public String getNarrator() { return this.narrator; }
  public double getLength() { return this.length; }
}
```

Sealed Interfaces

Analogous to `sealed` classes, `sealed` interfaces can also be defined. However, a `sealed` interface can have *both* subinterfaces and subclasses as its *permitted direct subtypes*. The permitted subtypes can be subclasses that *implement* the `sealed` interface and subinterfaces that *extend* the `sealed` interface. This is in contrast to a `sealed` class that can have direct subclasses, but not direct subinterfaces—as classes cannot be extended or implemented by interfaces.

Figure 5.10 shows the book domain from Figure 5.9 that has been augmented with a `sealed` interface that specifies its permitted direct subtypes in a `permits` clause:

Click here to view code image

```
public sealed interface Subscribable permits Ebook, Audiobook, VIPSubcribable
```

The `sealed` superinterface `Subscribable` has two `final` permitted direct subclasses (that implement the superinterface) and one `non-sealed` direct subinterface (that extends the superinterface). The declarations of the permitted direct subclasses `Ebook` and `Audiobook` have been updated accordingly so that they implement the direct superinterface `Subscribable`.

Click here to view code image

```
public final class Ebook extends Book implements Subscribable {}
public final class Audiobook extends Book implements Subscribable {}
public non-sealed interface VIPSubcribable extends Subscribable {}
```

The rest of the type declarations from Figure 5.9 remain the same in Figure 5.10:

Click here to view code image

```
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {}
public non-sealed class PrintedBook extends Book {}
```

Note that it is perfectly possible for a class or an interface to be a permitted direct subtype of more than one direct supertype—as is the case for the `Ebook` and the `Audiobook` subclasses in Figure 5.10.

Figure 5.10 *Sealed Classes and Interfaces*

We see from the discussion above that the permitted direct subtypes of a `sealed` superinterface abide by the same *contract* rules as for `sealed` superclasses:

- A permitted direct subclass or subinterface must extend or implement its direct superinterface, respectively.
- Any permitted subclass of a sealed interface must be declared either `sealed`, `non-sealed` or `final`, but any permitted subinterface can only be declared either `sealed` or `non-sealed`. The modifier `final` is not allowed for interfaces.
- The same rules for locality also apply for `sealed` interfaces and their permitted direct subtypes: All are declared either in the same named module or in the same package (named or unnamed) in the unnamed module.

Enum and Record Types as Permitted Direct Subtypes

By definition, an enum type (p. 287) is either *implicitly* `final` (has *no* enum constants that have a class body, as shown at (2)) or *implicitly* `sealed` (has *at least one* enum constant with a class body that constitutes an implicitly declared direct subclass, as shown at (3) for the constant `DVD_R` that has an empty class body). Thus an enum type can be specified as a permitted direct subtype of a `sealed` superinterface, as shown at (1). The modifiers `final` and `sealed` cannot be explicitly specified in the enum type declaration.

Click here to view code image

```
sealed interface MediaStorage permits CD, DVD {}       // (1) Sealed interface
enum CD implements MediaStorage {CD_ROM, CD_R, CD_W}   // (2) Implicitly final
enum DVD implements MediaStorage {DVD_R {}, DVD_RW}    // (3) Implicitly seale
```

Analogously, a record class (p. 299) is *implicitly* `final`, and can be specified as a permitted direct subtype of a `sealed` superinterface. The `sealed` interface `MediaStorage` at (1a) now permits the record class `HardDisk` as a direct subtype. Again note that the modifier `final` cannot be specified in the header of the `HardDisk` record class declared at (4).

Click here to view code image

```
sealed interface MediaStorage permits CD, DVD, HardDisk {}// (1a) Sealed inter
record HardDisk(double capacity) implements MediaStorage {}// (4) Implicitly f
```

Deriving Permitted Direct Subtypes of a Sealed Supertype

The `permits` clause of a `sealed` class or interface can be omitted, if all its permitted direct subtypes are declared in the *same compilation unit*—that is, are in the same source file. Since only one reference type can be declared `public` in a compilation unit, the accessibility of the other type declarations cannot be `public`.

The compiler can infer the permitted direct subtypes of any `sealed` class or interface in a compilation unit by checking whether a type declaration fulfills the contract for declaring a permitted direct subtype; that is, it extends/implements a `sealed` supertype and is declared either (implicitly) `sealed`, `non-sealed`, or (implicitly) `final`.

Click here to view code image

```
// File: Book.java (compilation unit)
```

```
public abstract sealed class Book {}        // Permitted subclasses are derive
non-sealed class PrintedBook extends Book {}

sealed interface Subscribable {}            // Permitted subtypes are derived.
final class Ebook extends Book implements Subscribable {}
final class Audiobook extends Book implements Subscribable {}
non-sealed interface VIPSubcribable extends Subscribable {}
```

Given the type declarations in the compilation unit `Book.java`, the compiler is able to infer that `PrintedBook`, `Ebook`, and `Audiobook` are permitted direct subclasses of the `sealed` superclass `Book`, whereas `Ebook`, `Audiobook`, and `VIPSubcribable` are permitted direct subtypes of the `sealed` superinterface `Subscribable`.

Using Sealed Classes and Interfaces

As we have seen, sealed classes and interfaces restrict which other classes or interfaces can extend or implement them. Apart from providing the programmer with a language feature to develop improved domain models for libraries, the compiler can also leverage the sealed types to provide better analysis of potential problems in the code at compile time.

Example 5.33 is a client that uses the `sealed` classes and their permitted direct subclasses defined in Example 5.32. Typically, a reference of the `sealed` superclass is used to process a collection of objects of its permitted subtypes. The array `books` is such a collection created at (2). The objects of the permitted subtypes are processed by the enhanced `for(:)` loop at (3). Typically, a cascading `if-else` statement is used in the loop body to distinguish each object in the collection and process it accordingly. Any object of a permitted subclass can be processed by the code as long as the cascading `if-else` statement guarantees that the checks are exhaustive—that is, all permitted subtypes are covered. The compiler cannot help to ensure that this is the case, as it cannot extract the necessary information from the cascading `if-else` statement for this analysis. Note that the conditionals in the cascading `if-else` statement use the `instanceof` pattern match operator to make the code at least less verbose.

Example 5.33 *Using Sealed Classes*

Click here to view code image

```
package signedandsealed;

public class BookAdmin {
  public static void main(String[] args) {
    // Create some books:                                                     //
    PrintedBook pBook = new PrintedBook("888-222", 340);
    Ebook eBook = new Ebook("999-777", "epub");
    Audiobook aBook = new Audiobook("333-555", "Fry", 300.0);

    // Create a book array:                                                   //
    Book[] books = {pBook, eBook, aBook};

    // Process the books:                                                     //
    for (Book book : books) {
      if (book instanceof PrintedBook pb) {                                   //
        System.out.printf("Printed book: %s, %d%n",
                           pb.getIsbn(), pb.getPageCount());
      } else if (book instanceof Ebook eb) {
        System.out.printf("Ebook: %s, %s%n", eb.getIsbn(), eb.getFormat());
      } else if (book instanceof Audiobook ab) {
        System.out.printf("Audiobook: %s, %s, %.1f%n",
                           ab.getIsbn(), ab.getNarrator(), ab.getLength());
      }
    }
  }
}
```

Output from the program:

Click here to view code image

```
Printed book: 888-222, 340
Ebook: 999-777, epub
Audiobook: 333-555, Fry, 300.0
```

Review Questions

5.32 What will be the result of attempting to compile and run the following code?

Click here to view code image

```
public enum Drill {
  ATTENTION("Attention!"), EYES_RIGHT("Eyes right!"),
  EYES_LEFT("Eyes left!"), AT_EASE("At ease!");

  private String command;

  Drill(String command) {
    this.command = command;
  }
  public static void main(String[] args) {
    System.out.println(ATTENTION);             // (1)
    System.out.println(AT_EASE);               // (2)
  }
}
```

Select the one correct answer.

a. The code compiles, but it reports a `ClassNotFoundException` when run, since an enum type cannot be run as a stand-alone application.

b. The compiler reports errors at (1) and (2), as the constants must be qualified by the enum type name `Drill`.

c. The compiler reports errors at (1) and (2), as the constants cannot be accessed in a static context.

d. The code compiles and prints:

```
ATTENTION
AT_EASE
```

e. The code compiles and prints:

```
Attention!
At ease!
```

f. None of the above

5.33 What will be the result of compiling and running the following code?

Click here to view code image

```
import java.util.Arrays;

public enum Priority {
  ONE(1) { public String toString() { return "LOW"; } },         // (1)
  TWO(2),
  THREE(3) { public String toString() { return "NORMAL"; } },    // (2)
  FOUR(4),
  FIVE(5) { public String toString() { return "HIGH"; } };       // (3)

  private int pValue;

  Priority(int pValue) {
    this.pValue = pValue;
  }

  public static void main(String[] args) {
    System.out.println(Arrays.toString(Priority.values()));
  }
}
```

Select the one correct answer.

a. The compiler reports syntax errors at (1), (2), and (3).

b. The code compiles and prints:
`[LOW, TWO, NORMAL, FOUR, HIGH]`

c. The code compiles and prints:
`[ONE, TWO, THREE, FOUR, FIVE]`

d. None of the above

5.34 What will be the result of compiling and running the following code?

Click here to view code image

```
public enum TrafficLight {
  RED("Stop"), YELLOW("Caution"), GREEN("Go");

  private String action;

  TrafficLight(String action) {
    this.action = action;
  }

  public static void main(String[] args) {
    TrafficLight green = new TrafficLight("Go");
    System.out.println(GREEN.equals(green));
  }
}
```

Select the one correct answer.

a. The code will compile and print `true`.

b. The code will compile and print `false`.

c. The code will fail to compile, as an enum type cannot be instantiated with the `new` operator.

d. The code will fail to compile, as the enum type does not provide the `equals()` method.

5.35 Given the following code:

Click here to view code image

```
public class RQ54 {
  public static void main(String[] args) {
    Product p = new Product(101, "Tea", 1.99);
    System.out.println(p);
  }
}
```

Which two record class definitions will cause this program to produce the following output?

Click here to view code image

```
Product[id=101, name=TEA, price=1.99]
```

Select the two correct answers.

a. Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product() {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

b. Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product(int newId, String newName, double newPrice) {
    id = newId;
    name = newName.toUpperCase();
    price = newPrice;
  }
}
```

c. Click here to view code image

```
public record Product() {
  public Product(int id, String name, double price) {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

d. Click here to view code image

```
public record Product(int id, String name, double price) {
  public Product {
    name = name.toUpperCase();
  }
}
```

e. Click here to view code image

```
public record Product(int id, String name, double price) {
  public String toString(){
    return "Product[id="+id+", name="+name.toUpperCase()+",
                price="+price+"]";
  }
}
```

f. Click here to view code image

```
public record Product(int id, String name, double price) {
 public String name() {
   return name.toUpperCase();
 }
 public String toString(){
   return "Product[id="+id+", name="+name+", price="+price+"]";
 }
}
```

5.36 Given the following code:

Click here to view code image

```
public record Employee(String name, double salary) { }
```

and

Click here to view code image

```
public class RQ55 {
  public static void main(String[] args) {
    Employee e1 = new Employee("Bob", 1234);
    Employee e2 = new Employee("Bob", 1234);
    System.out.print(e1.equals(e2));
    System.out.print(e1 == e2);
  }
}
```

What is the result?

Select the one correct answer.

a. `falsefalse`

b. `truetrue`

c. `falsetrue`

d. `truefalse`

5.37 Which statement is true?

Select the one correct answer.

a. A record class cannot declare new fields, other than the ones in its header.

b. A record class provides get and set methods for each component field.

c. A record class cannot have an `extends` clause.

d. A record class directly extends the class `Object`.

5.38 Which statement is true?

Select the one correct answer.

a. A class marked as `sealed` cannot be `abstract`.

b. A class marked as `non-sealed` cannot be `abstract`.

c. A class marked as `sealed` cannot be freely extended.

d. A class that extends a `sealed` class cannot be `sealed`.

e. A class that extends a `sealed` class cannot be `non-sealed`.

5.39 Given the following code:

Click here to view code image

```
public sealed interface X permits Y, Z { }
```

Which of the following options provides a correct definition of the permitted subtypes `Y` and `Z`?
Select the one correct answer.

a. Click here to view code image

```
public final class Y implements X { }
public sealed interface Z extends X { }
```

b. Click here to view code image

```
public final class Y implements X { }
public interface Z extends X { }
```

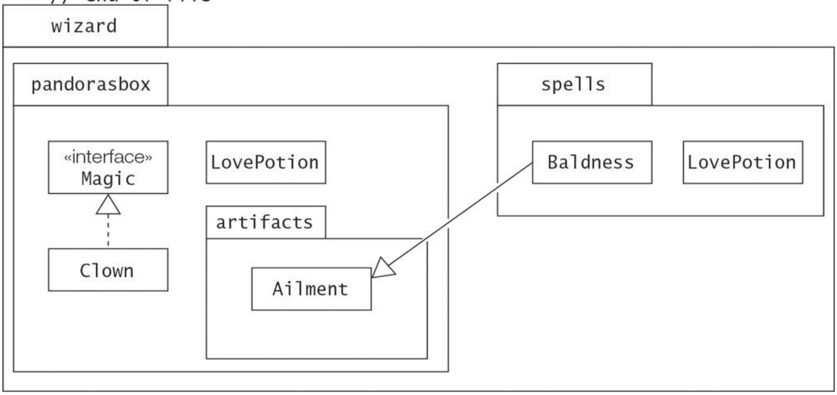
c. Click here to view code image

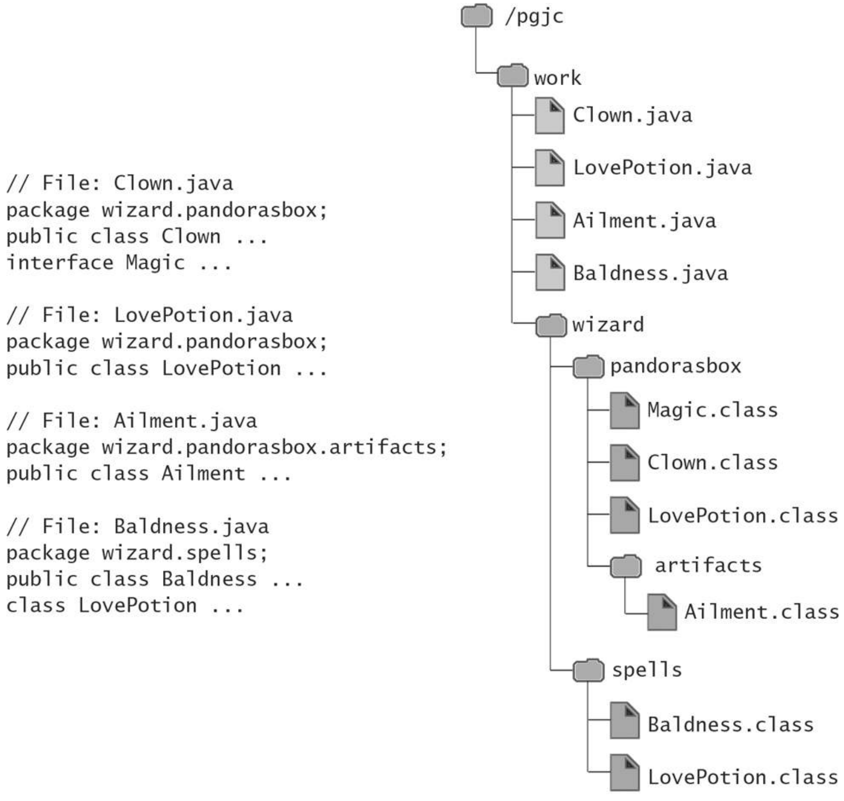
```
public non-sealed class Y implements X, Z { }
public sealed interface Z extends X permits Y { }
```

d. Click here to view code image

```
public class Y implements X, Z { }
public non-sealed interface Z extends X { }
```

```
// File: NewApp.java
// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilepackage;
// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;
import static java.lang.Math.*;
// PART 3: (ZERO OR MORE) top-level declarations
public class NewApp { }
class A { }
interface IX { }
record B() { }
enum C { }
// end of file
```

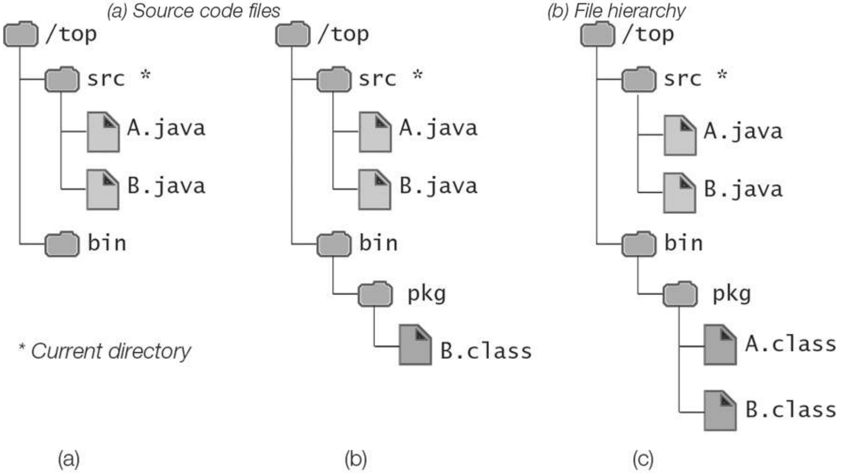

wizard
pandorasbox
«interface»
Magic
LovePotion
Clown
artifacts
Ailment
spells
Baldness
LovePotion


/pgjc
work
Clown.java
LovePotion.java
Ailment.java
Baldness.java
wizard
pandorasbox
Magic.class
Clown.class
LovePotion.class
artifacts
Ailment.class
spells
Baldness.class
LovePotion.class

```
// File: Clown.java
package wizard.pandorasbox;
public class Clown ...
interface Magic ...

// File: LovePotion.java
package wizard.pandorasbox;
public class LovePotion ...

// File: Ailment.java
package wizard.pandorasbox.artifacts;
public class Ailment ...

// File: Baldness.java
package wizard.spells;
public class Baldness ...
class LovePotion ...
```

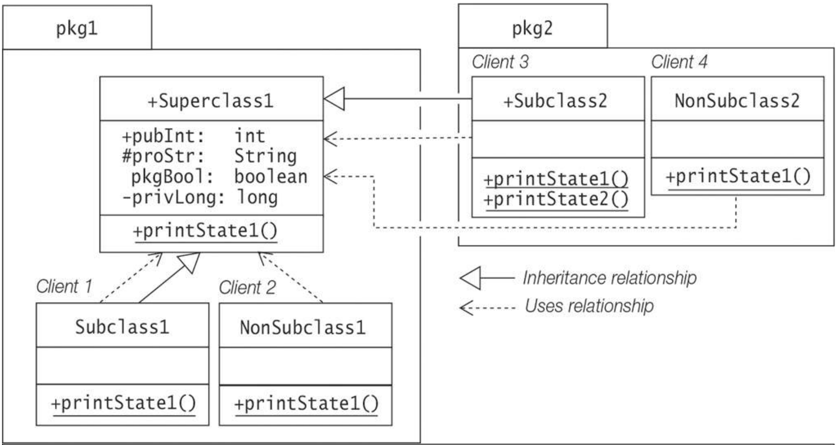
(a) Source code files

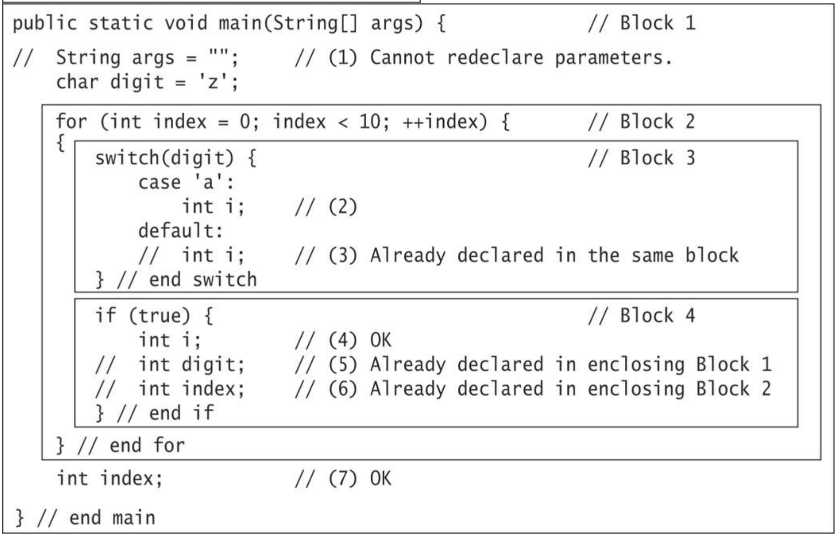
(b) File hierarchy


/top
src *
A.java
B.java
bin
* Current directory
(a)
/top
src *
A.java
B.java
bin
pkg
B.class
(b)
/top
src *
A.java
B.java
bin
pkg
A.class
B.class
(c)

```
// File: A.java
package pkg;
class A { B b; } // A uses B
```

```
// File: B.java
package pkg;
class B { }
```


pkg1
pkg2
Client 3
Client 4
+Superclass1
+pubInt: int
#proStr: String
pkgBool: boolean
-privLong: long
+printState1()
+Subclass2
+printState1()
+printState2()
NonSubclass2
+printState1()
Client 1
Client 2
Subclass1
+printState1()
NonSubclass1
+printState1()
Inheritance relationship
Uses relationship



```
public static void main(String[] args) {              // Block 1
//  String args = "";      // (1) Cannot redeclare parameters.
    char digit = 'z';

    for (int index = 0; index < 10; ++index) {        // Block 2
    {
        switch(digit) {                                // Block 3
            case 'a':
                int i;     // (2)
            default:
            //  int i;     // (3) Already declared in the same block
        } // end switch

        if (true) {                                    // Block 4
            int i;         // (4) OK
        //  int digit;     // (5) Already declared in enclosing Block 1
        //  int index;     // (6) Already declared in enclosing Block 2
        } // end if
    } // end for
    int index;             // (7) OK

} // end main
```

Access Control

6

Chapter Topics

- Understanding the features in Java that aid encapsulation
- Knowing the structure of a Java compilation unit
- Creating and using packages
- Using class-path to search for types during program compilation and execution
- Compiling and running code from packages
- Applying accessibility rules for top-level reference types and their members
- Applying scope rules for class members and local variables
- Implementing immutability

Java SE 17 Developer Exam O	
[3.4] Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable	§6.4, p. 324 §6.6, p. 352 §6.7, p. 356

○ *Variable scope, encapsulation, and immutability are covered in this chapter.*

○ *For local variable type inference, see §3.13, p. 142.*

Java SE 11 Developer Exam O	
[3.4] Understand variable scopes, applying encapsulation and make objects immutable	§6.4, p. 324 §6.6, p. 352 §6.7, p. 356

Java provides language features (such as classes, packages, and modules) that allow code to be encapsulated. This chapter covers creating and using packages, and compiling and running code from packages. Accessibility rules define how entities (e.g., classes and methods) are accessed in a Java program. Class scope rules define how class members are accessed in a class and block scope rules define how local variables are accessed in a block. Immutability of objects is a common class design strategy. The discussion in this chapter is based on *non-modular* code. Modules are discussed in Chapter 19, p. 1161.

6.1 Design Principle: Encapsulation

An object has properties and behaviors that are *encapsulated* inside the object. The services that the object offers to its clients make up its *contract*, or public interface. Only the contract defined by the object is available to the clients. The *implementation* of the object's properties and behavior is not a concern of the clients. Encapsulation helps to make clear the distinction between an object's contract and its implementation. This demarcation has major consequences for program development, as the implementation of an object can change without affecting the clients. Encapsulation also reduces complexity, as the internals of an object are hidden from the clients, which cannot alter its implementation.

Encapsulation is achieved through *information hiding*, by making judicious use of language features provided for this purpose. Information hiding in Java can be achieved at the following levels of granularity:

- *Method or block declaration*

  Localizing information in a method or a block hides it from the outside. Local variables can only be accessed in the method or the block according to their scope.

- *Reference type declaration*

  The accessibility of members declared in a reference type declaration (class, enum type, interface) can be controlled through access modifiers (p. 347). One much-advocated information-hiding practice is to prevent clients from having direct access to data maintained by an object. The fields of the object are `private`, and the object's contract defines `public` methods for the services provided by the object. Such tight encapsulation helps to separate the use from the implementation of a reference type.

- *Package declaration*

  Top-level reference types that belong together can be grouped into relevant packages by using the `package` statement. An `import` statement can be used to access types in other packages by their simple name. Inter-package accessibility of types can be controlled through `public` or package accessibility (p. 345).

- *Module declaration*

  Packages that are related can be grouped into a module by using a `module` declaration. How modules are created, accessed, compiled, and deployed is discussed in detail in Chapter 19, p. 1161.

Ample examples throughout the book illustrate how encapsulation can be achieved at different levels by using features provided by the Java programming language.

6.2 Java Source File Structure

The structure of a skeletal Java source file is depicted in Figure 6.1. A Java source file can have the following elements that, if present, must be specified in the following order:

1. An optional `package` declaration to specify a package name. Packages are discussed in §6.3, p. 326.
2. Zero or more `import` declarations. Since `import` declarations introduce type or static member names in the source code, they must be placed before any type declarations. Both type and static `import` declarations are discussed in §6.3, p. 329.
3. Any number of *top-level* type declarations. Class, enum, and interface declarations are collectively known as *type declarations*. Since these declarations belong to the same package, they are said to be defined at the *top level*, which is the package level.

   The type declarations can be defined in any order. Technically, a source file need not have any such declarations, but that is hardly useful.

   The JDK imposes the restriction that at most one `public` class declaration per source file can be defined. If a `public` class is defined, the file name must match this `public` class. For example, if the `public` class name is `NewApp`, the file name must be `NewApp.java`.

   Classes are discussed in §3.1, p. 99; interfaces are discussed in §5.6, p. 237; and enums are discussed in §5.13, p. 287.

Modules introduce another Java source file that contains a single module declaration (§19.3, p. 1168).

Note that except for the `package` and the `import` statements, all code is encapsulated in classes, interfaces, enums, and records. No such restriction applies to comments and whitespace.

Figure 6.1 *Java Source File Structure*

6.3 Packages

A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces, enums, and records.

Figure 6.2 shows an example of a package hierarchy comprising a package called `wizard` that contains two other packages: `pandorasbox` and `spells`. The package `pandorasbox` has a class called `Clown` that implements an interface called `Magic`, also found in the same package. In addition, the package `pandorasbox` has a class called `LovePotion` and a subpackage called `artifacts` containing a class called `Ailment`. The package `spells` has two classes: `Baldness` and `LovePotion`. The class `Baldness` is a subclass of class `Ailment` found in the subpackage `artifacts` in the package `pandorasbox`.

The dot (`.`) notation is used to uniquely identify package members in the package hierarchy. The class `wizard.pandorasbox.LovePotion`, for example, is different from the class `wizard.spells.LovePotion`. The `Ailment` class can be easily identified by the name `wizard.pandorasbox.artifacts.Ailment`, which is known as the *fully qualified name* of the type. Note that the fully qualified name of the type in a named package comprises the fully qualified name of the package and the simple name of the type. The *simple type name* `Ailment` and the *fully qualified package name* `wizard.pandorasbox.artifacts` together define the *fully qualified type name* `wizard.pandorasbox.artifacts.Ailment`.

Java programming environments usually map the fully qualified name of packages to the underlying (hierarchical) file system. For example, on a Unix system, the class file `LovePotion.class` corresponding to the fully qualified name `wizard.pandorasbox.LovePotion` would be found under the directory `wizard/pandorasbox`.

Figure 6.2 *Package Structure*

Conventionally, the *reverse* DNS (*Domain Name System*) notation based on the Internet domain names is used to uniquely identify packages. If the package `wizard` was implemented by a company called Sorcerers Limited that owns the domain `sorcerersltd.com`, its fully qualified name would be

```
com.sorcerersltd.wizard
```

Because domain names are unique, packages with this naming scheme are globally identifiable. It is not advisable to use the top-level package names `java` and `sun`, as these are reserved for the Java designers.

Note that each component of a package name must be a legal Java identifier. The following package would be illegal:

```
org.covid-19.2022.vaccine
```

The package name below is legal:

```
org.covid_19._2022.vaccine
```

A subpackage would be located in a subdirectory of the directory corresponding to its parent package. Apart from this locational relationship, a subpackage is an independent package with no other relation to its parent package. The subpackage `wizard.pandorasbox.artifacts` could easily have been placed elsewhere, as long as it was uniquely identified. Subpackages in a package do not affect the accessibility of the other package members. For all intents and purposes, subpackages are more an *organizational* feature than a language feature. Accessibility of members defined in type declarations is discussed in §6.5, p. 345.

Defining Packages

A package hierarchy represents an organization of the Java classes and interfaces. It does *not* represent the *source code* organization of the classes and interfaces. The source code is of no consequence in this regard. Each Java source file (also called *compilation unit*) can contain zero or more type declarations, but the compiler produces a separate *class* file containing the Java bytecode for each of them. A type declaration can indicate that its Java bytecode should be placed in a particular package, using a `package` declaration.

The `package` statement has the following syntax:

Click here to view code image

```
package fully_qualified_package_name;
```

At most, one `package` declaration can appear in a source file, and it must be the first statement in the source file. The package name is saved in the Java bytecode of the types contained in the package. Java naming conventions recommend writing package names in lowercase letters.

Note that this scheme has two consequences. First, all the classes and interfaces in a source file will be placed in the same package. Second, several source files can be used to specify the contents of a package.

If a `package` declaration is omitted in a compilation unit, the Java bytecode for the declarations in the compilation unit will belong to an *unnamed package* (also called the *default package*), which is typically synonymous with the current working directory on the host system.

Example 6.1 illustrates how the packages in Figure 6.2 can be defined using the `package` declaration. There are four compilation units. Each compilation unit has a `package` declaration, ensuring that the type declarations are compiled into the correct package. The complete code can be found in Example 6.7, p. 345.

Example 6.1 *Defining Packages and Using Type Import*

Click here to view code image

```
// File name: Clown.java                          // This file has 2 type declarati
package wizard.pandorasbox;                       // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing specific class

public class Clown implements Magic { /* ... */ }

interface Magic { /* ... */ }
```

Click here to view code image

```
// File name: LovePotion.java
package wizard.pandorasbox;                       // Package declaration

public class LovePotion { /* ... */ }
```

Click here to view code image

```
// File name: Ailment.java
package wizard.pandorasbox.artifacts;             // Package declaration

public class Ailment { /* ... */ }
```

Click here to view code image

```
// File name: Baldness.java                       // This file has 2 type declarati
package wizard.spells;                            // Package declaration

import wizard.pandorasbox.*;                      // (1) Type-import-on-demand
import wizard.pandorasbox.artifacts.*;            // (2) Import from subpackage

public class Baldness extends Ailment {           // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;           // (3) Fully qualified class name
  LovePotion tlcTwo;                              // Class in same package
  // ...
}
```

```
class LovePotion { /* ... */ }
```

Using Packages

The import facility in Java makes it easier to use the contents of packages. This subsection discusses importing *reference types* and *static members of reference types* from packages.

Importing Reference Types

The accessibility of types (classes, interfaces, and enums) in a package determines their access from other packages. Given a reference type that is accessible from outside a package, the reference type can be accessed in two ways. One way is to use the fully qualified name of the type. However, writing long names can become tedious. The second way is to use the `import` declaration that provides a shorthand notation for specifying the name of the type, often called *type import*.

The `import` declarations must be the first statement after any `package` declaration in a source file. The simple form of the `import` declaration has the following syntax:

Click here to view code image

```
import fully_qualified_type_name;
```

This is called *single-type-import*. As the name implies, such an `import` declaration provides a shorthand notation for a single type. The *simple* name of the type (i.e., its identifier) can now be used to access this particular type. Given the `import` declaration

Click here to view code image

```
import wizard.pandorasbox.Clown;
```

the simple name `Clown` can be used in the source file to refer to this class.

Alternatively, the following form of the `import` declaration can be used:

Click here to view code image

```
import fully_qualified_package_name.*;
```

This is called *type-import-on-demand*. It allows *any* type from the specified package to be accessed by its simple name. Given the `import` declaration

Click here to view code image

```
import wizard.pandorasbox.*;
```

the classes `Clown` and `LovePotion` and the interface `Magic` that are in the package `wizard.pandorasbox` can be accessed by their simple name in the source file.

An `import` declaration does not recursively import subpackages, as such nested packages are autonomous packages. The declaration also does not result in inclusion of the source code of the types; rather, it simply imports type names (i.e., it makes type names available to the code in a compilation unit).

All compilation units implicitly import the `java.lang` package (§8.1, p. 425). This is the reason why we can refer to the class `String` by its simple name, and need not use its fully qualified name `java.lang.String` all the time.

Import statements are *not* present in the compiled code, as all type names in the source code are replaced with their fully qualified names by the compiler.

Example 6.1 shows several usages of the `import` statement. Here we will draw attention to the class `Baldness` in the file `Baldness.java`. This class relies on two classes that have the same simple name `LovePotion` but are in different packages: `wizard.pandorasbox` and `wizard.spells`. To distinguish between the two classes, we can use their fully qualified names. However, since one of them is in the same package as the class `Baldness`, it is enough to fully qualify the class from the other package. This solution is used in Example 6.1 at (3). Note that the import of the `wizard.pandorasbox` package at (1) becomes redundant. Such name conflicts can usually be resolved by using variations of the `import` declaration together with fully qualified names.

The class `Baldness` extends the class `Ailment`, which is in the subpackage `artifacts` of the `wizard.pandorasbox` package. The `import` declaration at (2) is used to import the types from the subpackage `artifacts`.

The following example shows how a single-type-import declaration can be used to disambiguate a type name when access to the type is ambiguous by its simple name. The following `import` statement allows the simple name `List` to be used as shorthand for the `java.awt.List` type as expected:

Click here to view code image

```
import java.awt.*;           // imports all reference types from java.awt
```

Given the two `import` declarations

Click here to view code image

```
import java.awt.*;           // imports all type names from java.awt
import java.util.*;          // imports all type names from java.util
```

the simple name `List` is now ambiguous because both the types `java.util.List` and `java.awt.List` match.

Adding a single-type-import declaration for the `java.awt.List` type allows the simple name `List` to be used as a shorthand notation for this type:

Click here to view code image

```
import java.awt.*;           // imports all type names from java.awt
import java.util.*;          // imports all type names from java.util
import java.awt.List;        // imports the type List from java.awt explicitly
```

Importing Static Members of Reference Types

Analogous to the type import facility, Java also allows import of *static members* of reference types from packages, often called *static import*. Imported static members can be used by their simple names, and therefore need not be qualified. Importing static members of reference types from the unnamed package is not permissible.

The two forms of static import are shown here:

- *Single static import*: imports a specific static member from the designated type

  `import static` *fully_qualified_type_name*`.`*static_member_name*`;`

- *Static import on demand*: imports all static members in the designated type

  `import static` *fully_qualified_type_name*`.*;`

Both forms require the use of the keyword `import` followed by the keyword `static`, although the feature is called *static import*. In both cases, the *fully qualified name of the reference type* we are importing from is required.

The first form allows *single static import* of individual static members, and is demonstrated in Example 6.2. The constant `PI`, which is a static field in the class `java.lang.Math`, is imported at (1). Note the use of the fully qualified name of the type in the static import statement. The static method named `sqrt` from the class `java.lang.Math` is imported at (2). Only the *name* of the static method is specified in the static import statement; no parameters are listed. Use of any other static member from the `Math` class requires that the fully qualified name of the class be specified. Since types from the `java.lang` package are imported implicitly, the fully qualified name of the `Math` class is not necessary, as shown at (3).

Static import on demand is easily demonstrated by replacing the two `import` statements in Example 6.2 with the following `import` statement:

Click here to view code image

```
import static java.lang.Math.*;
```

We can also dispense with the use of the class name `Math` at (3), as all static members from the `Math` class are now imported:

Click here to view code image

```
double hypotenuse = hypot(x, y);   // (3') Type name can now be omitted.
```

Example 6.2 *Single Static Import*

Click here to view code image

```
import static java.lang.Math.PI;           // (1) Static field
import static java.lang.Math.sqrt;         // (2) Static method
// Only specified static members are imported.

public class Calculate3 {
  public static void main(String[] args) {
    double x = 3.0, y = 4.0;

    double squareroot = sqrt(y);           // Simple name of static method
    double hypotenuse = Math.hypot(x, y);  // (3) Requires type name
    double area = PI * y * y;              // Simple name of static field
    System.out.printf("Square root: %.2f, hypotenuse: %.2f, area: %.2f%n",
                        squareroot, hypotenuse, area);
  }
}
```

Output from the program:

Click here to view code image

```
Square root: 2.00, hypotenuse: 5.00, area: 50.27
```

Example 6.3 illustrates how static import can be used to access interface constants (§5.6, p. 254). The static `import` statement at (1) allows the interface constants in the package `mypkg` to be accessed by their simple names. The static import facility avoids the `MyFactory` class having to *implement* the interface so as to access the constants by their simple name (often referred to as the *interface constant antipattern*):

Click here to view code image

```
public class MyFactory implements mypkg.IMachineState {
 // ...
}
```

Example 6.3 *Avoiding the Interface Constant Antipattern*

Click here to view code image

```
package mypkg;

public interface IMachineState {
  // Fields are public, static, and final.
  int BUSY = 1;
  int IDLE = 0;
  int BLOCKED = -1;
}
```

Click here to view code image

```
import static mypkg.IMachineState.*;    // (1) Static import interface constan

public class MyFactory {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED }; // (2) Access by simple name
    for (int s : states)
      System.out.print(s + " ");
  }
}
```

Output from the program:

```
0 1 0 -1
```

Static import is ideal for importing enum constants from packages, as such constants are static members of an enum type (§5.13, p. 287). Example 6.4 combines type and static imports. The enum constants can be accessed at (5) using their simple names because of the static import statement at (2). The type import at (1) is required to access the enum type `State` by its simple name at (4) and (6).

Example 6.4 *Importing Enum Constants*

Click here to view code image

```
package mypkg;

public enum State { BUSY, IDLE, BLOCKED }
```

Click here to view code image

```
// File: Factory.java (in unnamed package)
import mypkg.State;                   // (1) Single type import

import static mypkg.State.*;          // (2) Static import on demand
import static java.lang.System.out;   // (3) Single static import

public class Factory {
  public static void main(String[] args) {
    State[] states = {                // (4) Using type import implied by (1)
        IDLE, BUSY, IDLE, BLOCKED     // (5) Using static import implied by (2)
    };
    for (State s : states)            // (6) Using type import implied by (1)
      out.print(s + " ");             // (7) Using static import implied by (3)
  }
}
```

Output from the program:

```
IDLE BUSY IDLE BLOCKED
```

Identifiers in a class can *shadow* static members that are imported. Example 6.5 illustrates the case where the parameter `out` of the method `writeInfo()` has the same name as the statically imported field `java.lang.System.out`. The type of the parameter `out` is `ShadowImport` and that of the statically imported field `out` is `PrintStream`. Both classes `PrintStream` and `ShadowImport` define the method `println()` that is called in the program. The only way to access the imported field `out` in the method `write-Info()` is to use its fully qualified name.

Example 6.5 *Shadowing Static Import*

Click here to view code image

```
import static java.lang.System.out;       // (1) Static import

public class ShadowImport {

  public static void main(String[] args) {
    out.println("Calling println() in java.lang.System.out");
    ShadowImport sbi = new ShadowImport();
    writeInfo(sbi);
  }

  // Parameter out shadows java.lang.System.out:
  public static void writeInfo(ShadowImport out) {
    out.println("Calling println() in the parameter out");
    System.out.println("Calling println() in java.lang.System.out"); // Qualif
  }

  public void println(String msg) {
    out.println(msg + " of type ShadowImport");
  }
}
```

Output from the program:

Click here to view code image

```
Calling println() in java.lang.System.out
Calling println() in the parameter out of type ShadowImport
Calling println() in java.lang.System.out
```

The next code snippet illustrates a common conflict that occurs when a static field with the same name is imported by *several* static import statements. This conflict is readily resolved by using the fully qualified name of the field. In the case shown here, we can use the simple name of the class in which the field is declared, as the `java.lang` package is implicitly imported by all compilation units.

Click here to view code image

```
import static java.lang.Integer.MAX_VALUE;
import static java.lang.Double.MAX_VALUE;

public class StaticFieldConflict {
  public static void main(String[] args) {
    System.out.println(MAX_VALUE);          // (1) Ambiguous! Compile-time err
    System.out.println(Integer.MAX_VALUE);  // OK
    System.out.println(Double.MAX_VALUE);   // OK
  }
}
```

Conflicts can also occur when a static method with the same signature is imported by several static import statements. In Example 6.6, a method named `binarySearch` is imported 21 times by the static import statements. This method is overloaded twice in the `java.util.Collections` class and 18 times in the `java.util.Arrays` class, in addition to one declaration in the `mypkg.Auxiliary` class. The classes `java.util.Arrays` and `mypkg.Auxiliary` have a declaration of this method with the *same signature* (`binarySearch(int[], int)`) that matches the method call at (2), resulting in a signature conflict that is flagged as a compile-time error. The conflict can again be resolved by specifying the fully qualified name of the method.

If the static import statement at (1) is removed, there is no conflict, as only the class `java.util.Arrays` has a method that matches the method call at (2). If the declaration of the method `binarySearch()`

at (3) is allowed, there is also *no* conflict, as this method declaration will *shadow* the imported method whose signature it matches.

Example 6.6 *Conflict in Importing a Static Method with the Same Signature*

Click here to view code image

```
package mypkg;

public class Auxiliary {
  public static int binarySearch(int[] a, int key) { // Same in java.util.Arra
    // Implementation is omitted.
    return -1;
  }
}
```

Click here to view code image

```
// File: MultipleStaticImport.java (in unnamed package)
import static java.util.Collections.binarySearch;  //    2 overloaded methods
import static java.util.Arrays.binarySearch;       // + 18 overloaded methods
import static mypkg.Auxiliary.binarySearch; // (1) Causes signature conflict

public class MultipleStaticImport {
  public static void main(String[] args) {
    int index = binarySearch(new int[] {10, 50, 100}, 50); // (2) Ambiguous!
    System.out.println(index);
  }

//public static int binarySearch(int[] a, int key) {      // (3)
//  return -1;
//}
}
```

Compiling Code into Package Directories

Conventions for specifying pathnames vary on different platforms. In this chapter, we will use pathname conventions used on a Unix-based platform. While trying out the examples in this section, attention should be paid to platform dependencies in this regard—especially the fact that the *separator characters* in *file paths* for the Unix-based and Windows platforms are `/` and `\`, respectively.

As mentioned earlier, a package can be mapped on a hierarchical file system. We can think of a package name as a pathname in the file system. Referring to Example 6.1, the package name `wizard.pandorasbox` corresponds to the pathname `wizard/pandorasbox`. The Java bytecode for all types declared in the source files `Clown.java` and `LovePotion.java` will be placed in the *package directory* with the pathname `wizard/pandorasbox`, as these source files have the following `package` declaration:

```
package wizard.pandorasbox;
```

The *location* in the file system where the package directory should be created is specified using the `-d` option (`d` for *destination*) of the `javac` command. The term *destination directory* is a synonym for this location in the file system. The compiler will create the package directory with the pathname `wizard/pandorasbox` (including any subdirectories required) *under* the specified location, and will place the Java bytecode for the types declared in the source files `Clown.java` and `LovePotion.java` inside the package directory.

Assuming that the current directory (`.`) is the directory `/pgjc/work`, and the four source code files in Figure 6.3a (see also Example 6.1) are found in this directory, the following command issued in the current directory will create a file hierarchy (Figure 6.3b) under this directory that mirrors the package structure in Figure 6.2, p. 327:

Click here to view code image

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
```

Note that two of the source code files in Figure 6.3a have multiple type declarations. Note also the subdirectories that are created for a fully qualified package name, and where the class files are located. In this command line, the space between the `-d` option and its argument is mandatory.

Figure 6.3 *Compiling Code into Package Directories*

The wildcard `*` can be used to specify all Java source files to be compiled from a directory. It expands to the names of the Java source files in that directory. The two commands below are equivalent to the command above.

```
>javac -d . *.java
>javac -d . ./*.java
```

We can specify any *relative* pathname that designates the destination directory, or its *absolute* pathname:

Click here to view code image

```
>javac -d /pgjc/work Clown.java LovePotion.java Ailment.java Baldness.java
```

We can, of course, specify destinations other than the current directory where the class files with the bytecode should be stored. The following command in the current directory `/pgjc/work` will create the necessary packages with the class files under the destination directory `/pgjc/myapp`:

Click here to view code image

```
>javac -d ../myapp Clown.java LovePotion.java Ailment.java Baldness.java
```

Without the `-d` option, the default behavior of the `javac` compiler is to place all class files directly under the current directory (where the source files are located), rather than in the appropriate subdirectories corresponding to the packages.

The compiler will report an error if there is any problem with the destination directory specified with the `-d` option (e.g., if it does not exist or does not have the right file permissions).

Running Code from Packages

Referring to Example 6.1, if the current directory has the absolute pathname `/pgjc/ work` and we want to run `Clown.class` in the directory with the pathname `./wizard/ pandorasbox`, the *fully qualified name* of the `Clown` class *must* be specified in the `java` command:

Click here to view code image

```
>java wizard.pandorasbox.Clown
```

This command will load the bytecode of the class `Clown` from the file with the pathname `./wizard/pandorasbox/Clown.class`, and will start the execution of its `main()` method.

6.4 Searching for Classes on the Class Path

A program typically uses other precompiled classes and libraries, in addition to the ones provided by the Java standard libraries. In order for the JDK tools to find these files efficiently, the `CLASSPATH` environment variable or the `-classpath` option can be used, both of which are explained below.

In particular, the `CLASSPATH` environment variable can be used to specify the *class search path* (usually abbreviated to just *class path*), which is the *pathnames* or *locations* in the file system where JDK tools should look when searching for third-party and user-defined classes. Alternatively, the `-classpath` option (*short form* `-cp`) of the JDK tool commands can be used for the same purpose. The `CLASSPATH` environment variable is not recommended for this purpose, as its class path value affects *all* Java applications on the host platform, and any application can modify it. However, the `-classpath` option can be used to set the class path for each application individually. This way, an application cannot modify the class path for other applications. The class path specified in the `-classpath` option supersedes the path or paths set by the `CLASSPATH` environment variable while the JDK tool command is running. We will not discuss the `CLASSPATH` environment variable here, and will assume it to be undefined.

Basically, the JDK tools first look in the directories where the Java standard libraries are installed. If the class is not found in the standard libraries, the tool searches in the class path. When no class path is defined, the default value of the class path is assumed to be the current directory. If the `-classpath` option is used and the current directory should be searched by the JDK tool, the current directory must be specified as an entry in the class path, just like any other directory that should be searched. This is most conveniently done by including `'.'` as one of the entries in the class path.

We will use the file hierarchies shown in Figure 6.4 to illustrate some of the intricacies involved when searching for classes. The current directory has the absolute pathname `/top/src`, where the source files are stored. The package `pkg` will be created under the directory with the absolute pathname `/top/bin`. The source code in the two source files `A.java` and `B.java` is also shown in Figure 6.4.

Figure 6.4 *Searching for Classes*

The file hierarchy before any files are compiled is shown in Figure 6.4a. Since the class `B` does not use any other classes, we compile it first with the following command, resulting in the file hierarchy shown in Figure 6.4b:

```
>javac -d ../bin B.java
```

Next, we try to compile the file `A.java`, and we get the following results:

Click here to view code image

```
>javac -d ../bin A.java
A.java:3: cannot find symbol
symbol  : class B
location: class pkg.A
```

```
public class A { B b; }
                 ^
1 error
```

The compiler cannot find the class `B`—that is, the file `B.class` containing the Java bytecode for the class `B`. In Figure 6.4b, we can see that it is in the package `pkg` under the directory `bin`, but the compiler cannot find it. This is hardly surprising, as there is no bytecode file for the class `B` in the current directory, which is the default value of the class path. The following command sets the value of the class path to be `/top/bin`, and compilation is successful (Figure 6.4c):

Click here to view code image

```
>javac -classpath /top/bin -d ../bin A.java
```

It is very important to understand that when we want the JDK tool to search in a *named package*, it is the *location* or the *root* of the package that is specified; in other words, the class path indicates the directory that *contains* the first component of the fully qualified package name. In Figure 6.4c, the package `pkg` is contained under the directory whose absolute path is `/top/bin`. The following command will *not* work, as the directory `/top/bin/pkg` does *not* contain a package with the name `pkg` that has a class `B`:

Click here to view code image

```
>javac -classpath /top/bin/pkg -d ../bin A.java
```

Also, the compiler is *not* using the class path to find the source file(s) that are specified in the command line. In the preceding command, the source file has the relative pathname `./A.java`. Consequently, the compiler looks for the source file in the current directory. The class path is used to find the classes used by the class `A`, in this case, to find class `B`.

Given the file hierarchy in Figure 6.3, the following `-classpath` option sets the class path so that *all* packages (`wizard.pandorasbox`, `wizard.pandorasbox.artifacts`, `wizard.spells`) in Figure 6.3 will be searched, as all packages are located under the specified directory:

```
-classpath /pgjc/work
```

However, the following `-classpath` option will not help in finding *any* of the packages in Figure 6.3, as none of the *packages* are located under the specified directory:

Click here to view code image

```
>java -classpath /pgjc/work/wizard pandorasbox.Clown
```

This command also illustrates an important point about package names: The *fully qualified package name* should not be split. The package name for the class `wizard.pandorasbox.Clown` is `wizard.pandorasbox`, and must be specified fully. The following command will search all packages in Figure 6.3 for classes that are used by the class `wizard.pandorasbox.Clown`:

Click here to view code image

```
>java -classpath /pgjc/work wizard.pandorasbox.Clown
```

The class path can specify several *entries* (i.e., several locations), and the JDK tool searches them in the order they are specified, from left to right.

Click here to view code image

```
-classpath /pgjc/work:/top/bin:.
```

We have used the path-separator character `':'` for Unix-based platforms to separate the entries, and also included the current directory (`.`) as an entry. There should be no whitespace on either side of the path-separator character. On the Windows platform, the path-separator character is a semicolon (`;`).

The search in the class path entries stops once the required class file is found. Therefore, the order in which entries are specified can be significant. If a class `B` is found in a package `pkg` located under the directory `/ext/lib1`, and also in a package `pkg` located under the directory `/ext/lib2`, the order in which the entries are specified in the two `-classpath` options shown below is significant. They will result in the class `pkg.B` being found under `/ext/lib1` and `/ext/lib2`, respectively.

Click here to view code image

```
-classpath /ext/lib1:/ext/lib2
-classpath /ext/lib2:/ext/lib1
```

The examples so far have used absolute pathnames for class path entries. We can, of course, use relative pathnames as well. If the current directory has the absolute pathname `/pgjc/work` in Figure 6.3, the following command will search the packages under the current directory:

Click here to view code image

```
>java -classpath . wizard.pandorasbox.Clown
```

If the current directory has the absolute pathname `/top/src` in Figure 6.4, the following command will compile the file `./A.java`:

Click here to view code image

```
>javac -classpath ../bin -d ../bin A.java
```

If the name of an entry in the class path includes whitespace, the name should be double-quoted so that it will be interpreted correctly:

```
-classpath "../new bin"
```

Table 6.1 summarizes the commands and options to compile and execute non-modular code. The tool documentation from Oracle provides more details on how to use the JDK development tools.

Table 6.1 *Compiling and Executing Non-Modular Java Code*

Operation	Command
Compiling non-modular code	`javac --class-path` *classpath* `-d` *directory sourceFiles* `javac -classpath` *classpath* `-d` *directory sourceFiles* `javac -cp` *classpath* `-d` *directory sourceFiles*
Executing non-modular code	`java --class-path` *classpath qualifiedClassName* `java -classpath` *classpath qualifiedClassName* `java -cp` *classpath qualifiedClassName*

Review Questions

6.1 Which of the following statements are true about the following code in a source file?

Click here to view code image

```
package net.alphabet;
import java.util.ArrayList;
public class A {}
class B {}
```

Select the two correct answers.

a. Both class `A` and class `B` will be placed in the package `net.alphabet`.

b. Only class `A` will be placed in the package `net.alphabet`. Class `B` will be placed in the default package.

c. Both class `A` and class `B` can access the imported class `java.util.ArrayList` by its simple name.

d. Only class `A` can access the imported class `java.util.ArrayList` by its simple name.

6.2 Given the following code:

Click here to view code image

```
package app;
public class Window {
  final static String frame = "Top-frame";
}
```

Click here to view code image

```
package app;
// (1) INSERT IMPORT STATEMENT HERE
public class Canvas {
  private String str = frame;
}
```

Which import statement, when inserted at (1), will make the code compile?
Select the one correct answer.

a. `import app.*;`

b. `import app.Window;`

c. `import java.lang.*;`

d. `import java.lang.String;`

e. `import static app.Window.frame;`

6.3 Given the following code:

Click here to view code image

```
package mainpkg.subpkg1;
public class Window {}
```

Click here to view code image

```
package mainpkg.subpkg2;
public class Window {}
```

Click here to view code image

```
package mainpkg;
// (1) INSERT IMPORT STATEMENTS HERE
public class Screen {
  private Window win;
}
```

Which import statement, when inserted independently at (1),

will make the code compile?
Select the four correct answers.

a. `import mainpkg.*;`

b. `import mainpkg.subpkg1.*;`

c. `import mainpkg.subpkg2.*;`

d. Click here to view code image

```
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.Window;
```

e. Click here to view code image

```
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.*;
```

f. Click here to view code image

```
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.*;
```

g. Click here to view code image

```
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.Window;
```

6.4 Given the following code:

Click here to view code image

```
// (1) INSERT ONE IMPORT STATEMENT HERE
public class RQ700A20 {
  public static void main(String[] args) {

    System.out.println(sqrt(49));
  }
}
```

Which import statement, when inserted independently at (1), will make the program print `7` when the program is compiled and run?
Select the two correct answers.

a. `import static Math.*;`

b. `import static Math.sqrt;`

c. `import static java.lang.Math.sqrt;`

d. `import static java.lang.Math.sqrt();`

e. `import static java.lang.Math.*;`

6.5 Given the following directory structure:

Click here to view code image

```
/top
  |--- wrk
        |--- pkg
              |--- A.java
              |--- B.java
```

assume that the two files `A.java` and `B.java` contain the following code, respectively:

```
package pkg;
class A { B b; }
```

```
package pkg;
class B {}
```

For which combinations of current directory and command is the compilation successful?
Select the two correct answers.

a. Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp .:pkg A.java
```

b. Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp . pkg/A.java
```

c. Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp pkg A.java
```

d. Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp .:pkg pkg/A.java
```

e. Click here to view code image

```
Current directory: /top/wrk/pkg
Command: javac A.java
```

f. Click here to view code image

```
Current directory: /top/wrk/pkg
Command: javac -cp . A.java
```

6.6 Given the following code:

Click here to view code image

```
package a.b;
public class X {
  public static int y = 100;
}
```

Click here to view code image

```
package a.b.c;
// (1) INSERT IMPORT STATEMENT HERE
public class Z {
  public void xyz() {

    int v = y;
  }
}
```

Which import statement, when inserted individually at (1), will allow class `Z` to compile?
Select the two correct answers.

a. `import a.b.*;`
b. `import a.b.X.*;`
c. `import static a.b.*;`
d. `import static a.b.X.*;`
e. `import static a.b.X.y;`

6.7 Given the following code:

Click here to view code image

```
package life.animals;
public class Cat { }
```

Click here to view code image

```
package life.animals;
public class Cow { }
```

Click here to view code image

```
package life.animals;
public class Dog { }
```

Click here to view code image

```
package habitat;
// (1) INSERT IMPORT STATEMENTS HERE
public class Farm {
  private Cat cat;
  private Cow cow;
}
```

Which import statements, when inserted individually at (1), will allow class `Farm` to compile?
Select the two correct answers.

a. `import life.animals.*;`
b. `import static life.animals.*;`
c. Click here to view code image

```
import static life.animals.Cat;
import static life.animals.Cow;
```

d. `import life.animals.Cat;`
`import life.animals.Cow;`

6.8 Which statement is true about `import` statements?

Select the one correct answer.

a. Import of a package also imports all subpackages.

b. Import of a package with a wildcard also imports all subpackages.

c. Import statements are required to access code in other packages.

d. All import statements are removed by the compiler.

6.5 Access Modifiers

In this section, we discuss accessibility of top-level type declarations that can be encapsulated into packages and accessibility of members that can be encapsulated in a top-level type declaration. A top-level reference type is a reference type (class, interface, enum, record) that is not declared inside another reference type.

Access modifiers are sometimes also called *visibility modifiers*.

Access Modifiers for Top-Level Type Declarations

The access modifier `public` can be used to declare top-level reference types that are accessible from everywhere, both from inside their own package and from inside other packages. If the access modifier is omitted, the reference types can be accessed only in their own package and not in any other packages—that is, they have *package access*, also called *package-private* or *default access*.

The packages shown in Figure 6.2, p. 327, are implemented by the code in Example 6.7. Class files with Java bytecode for top-level type declarations are placed in designated packages using the `package` statement. A top-level type declaration from one package can be accessed in another packages either by using the *fully qualified name* of the type or by using an `import` statement to import the type so that it can be accessed by its *simple name*.

Example 6.7 *Access Modifiers for Top-Level Reference Types*

Click here to view code image

```
// File: Clown.java
package wizard.pandorasbox;                  // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing class Ailment

public class Clown implements Magic {        // (1)
  LovePotion tlc;                            // Class in same package
  Ailment problem;                           // Simple class name
  Clown() {
    tlc = new LovePotion("passion");
    problem = new Ailment("flu");            // Simple class name
  }
  @Override public void levitate()  {        // (2)
    System.out.println("Levitating");
  }
  public void mixPotion()   { System.out.println("Mixing " + tlc); }
  public void healAilment() { System.out.println("Healing " + problem); }

  public static void main(String[] args) {
    Clown joker = new Clown();
    joker.levitate();
    joker.mixPotion();
    joker.healAilment();
  }
}

interface Magic { void levitate(); }         // (3)
```

Click here to view code image

```
// File: LovePotion.java
package wizard.pandorasbox;                  // Package declaration

public class LovePotion {                    // (4) Accessible outside package
  String potionName;
  public LovePotion(String name) { potionName = name; }
  public String toString()       { return potionName; }
}
```

Click here to view code image

```
// File: Ailment.java
package wizard.pandorasbox.artifacts;        // Package declaration

public class Ailment {                       // Accessible outside package
  String ailmentName;
  public Ailment(String name) { ailmentName = name; }
  public String toString() { return ailmentName; }
}
```

Click here to view code image

```
// File: Baldness.java
package wizard.spells;                       // Package declaration

import wizard.pandorasbox.*;                 // Redundant
import wizard.pandorasbox.artifacts.*;       // Import of subpackage

public class Baldness extends Ailment {      // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;      // Fully qualified name
  LovePotion tlcTwo;                         // Class in same package
  Baldness(String name) {
    super(name);
    tlcOne = new wizard.pandorasbox.         // Fully qualified name
                 LovePotion("romance");
    tlcTwo = new LovePotion();               // Class in same package
  }
}

class LovePotion /* implements Magic */ {    // (5) Magic is not accessible
  // @Override public void levitate() {}     // (6) Cannot override method
}
```

Compiling and running the program from the current directory gives the following results:

Click here to view code image

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
>java wizard.pandorasbox.Clown
Levitating
Mixing passion
Healing flu
```

In Example 6.7, the class `Clown` at (1) and the interface `Magic` at (3) are placed in a package called `wizard.pandorasbox`. The `public` class `Clown` is accessible from everywhere. The `Magic` interface has package accessibility, and can only be accessed within the package `wizard.pandorasbox`. It is not accessible from other packages, not even from subpackages.

The class `LovePotion` at (4) is also placed in the package called `wizard.pandorasbox`. The class has `public` accessibility, and is therefore accessible from other packages. The two files `Clown.java` and `LovePotion.java` demonstrate how several compilation units can be used to group classes in the same package, as the type declarations in these two source files are placed in the package `wizard.pandorasbox`.

In the file `Clown.java`, the class `Clown` at (1) implements the interface `Magic` at (3) from the same package. We have used the annotation `@Override` in front of the declaration of the `levitate()` method at (2) so that the compiler can aid in checking that this method is declared correctly as required by the interface `Magic`.

In the file `Baldness.java`, the class `LovePotion` at (5) wishes to implement the interface `Magic` at (3) from the package `wizard.pandorasbox`, but this is not possible, although the source file imports from this package. The reason is that the interface `Magic` has package accessibility, and can therefore only be accessed within the package `wizard.pandorasbox`. The method `levitate()` of the `Magic` interface therefore cannot be overridden in class `LovePotion` at (6).

Table 6.2 summarizes accessibility of top-level reference types in a package. Just because a reference type is accessible does not necessarily mean that members of the type are also accessible. Accessibility of members is governed separately from type accessibility, as explained in the next subsection.

Table 6.2 *Access Modifiers for Top-Level Reference Types (Non-Modular)*

Modifiers	Top-level types

No modifier	Accessible in its own package (*package accessibility*)
`public`	Accessible anywhere

Access Modifiers for Class Members

By specifying member access modifiers, a class can control which information is accessible to clients (i.e., other classes). These modifiers help a class define a *contract* so that clients know exactly which services are offered by the class.

The accessibility of a member in a class can be any one of the following:

- `public`
- `protected`
- *package access* (also known as *package-private* and *default access*), when no access modifier is specified
- `private`

In the following discussion of access modifiers for members of a class, keep in mind that the member access modifier has meaning only if the class (or one of its subclasses) is accessible to the client. Also, note that only one access modifier can be specified for a member.

The discussion in this subsection applies to both instance and static members of top-level classes.

In UML notation, when applied to member names the prefixes `+`, `#`, and `-` indicate `public`, `protected`, and `private` member access, respectively. No access modifier indicates package access for class members.

The package hierarchy shown in Figure 6.5 is implemented by the code in Example 6.8. The class `Superclass1` at (1) in `pkg1` has two subclasses: `Subclass1` at (3) in `pkg1` and `Subclass2` at (5) in `pkg2`. The class `Superclass1` in `pkg1` is used by the other classes (designated as Client 1 to Client 4) in Figure 6.5.

Figure 6.5 *Accessibility of Class Members*

Accessibility of a member is illustrated in Example 6.8 by the four instance fields defined in the class `Superclass1` in `pkg1`, where each instance field has a different accessibility. These four instance fields are accessed in a `Superclass1` object created in the `static` method `printState1()` declared in five different contexts:

- Defining class: The class in which the member is declared—that is, `pkg1.Superclass1` in which the four instance fields being accessed are declared
- Client 1: From a subclass in the same package—that is, `pkg1.Subclass1`
- Client 2: From a non-subclass in the same package—that is, `pkg1.NonSubclass1`
- Client 3: From a subclass in another package—that is, `pkg2.Subclass2`
- Client 4: From a non-subclass in another package—that is, `pkg2.NonSubclass2`

Example 6.8 *Accessibility of Class Members*

Click here to view code image

```
// File: Superclass1.java
package pkg1;

import static java.lang.System.out;

public class Superclass1 {                              // (1)

  // Instance fields with different accessibility:
  public    int     pubInt   = 2017;
  protected String  proStr   = "SuperDude";
            boolean pgkBool  = true;
  private   long    privLong = 0x7777;

  public static void printState1() {                    // (2)
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);
  }
}

// Client 1
class Subclass1 extends Superclass1 {                   // (3)
  public static void printState1() {

    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}

// Client 2
class NonSubclass1 {                                    // (4)
  public static void printState1() {

    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}
```

Click here to view code image

```
// File: Subclass2.java
package pkg2;
import pkg1.Superclass1;

import static java.lang.System.out;

// Client 3
public class Subclass2 extends Superclass1 {     // (5)

  public static void printState1() {             // (6)
    Superclass1 obj1 = new Superclass1();        // Object of Superclass1
    out.println(obj1.pubInt);
    out.println(obj1.proStr);   // (7) Compile-time error! Protected access.
    out.println(obj1.pgkBool);  // Compile-time error! Package access.
    out.println(obj1.privLong); // Compile-time error! Private access.
  }

  public static void printState2() {             // (8)
    Subclass2 obj2 = new Subclass2();            // (9) Object of Subclass2
    out.println(obj2.pubInt);
    out.println(obj2.proStr);   // (10) OK! Protected access.
    out.println(obj2.pgkBool);  // Compile-time error! Package access.
    out.println(obj2.privLong); // Compile-time error! Private access.
  }
}

// Client 4
class NonSubclass2 {                             // (11)
  public static void printState1() {
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);   // Compile-time error! Protected access.
    out.println(obj1.pgkBool);  // Compile-time error! Package access.
    out.println(obj1.privLong); // Compile-time error! Private access.
  }
}
```

`public` *Members*

Public access is the least restrictive of all the access modifiers. A `public` member is accessible from anywhere, both in the package containing its class and by other packages where its class is accessible.

In Example 6.8, the `public` instance field `pubInt` in an instance of the class `Superclass1` is accessible by all four clients. Subclasses can access their inherited `public` members by their simple names, and all clients can access `public` members in an instance of the class `Superclass1`.

`protected` *Members*

A `protected` member is accessible in all classes in the same package, and by all subclasses of its class in any package where its class is accessible.

In other words, a `protected` member cannot be accessed by non-subclasses in other packages. This kind of access is more restrictive than `public` member access.

In Example 6.8, the `protected` instance field `proStr` in an instance of the class `Superclass1` is accessible within `pkg1` by Client 1 and Client 2. Also as expected, Client 4—the class `NonSubclass2` in `pkg2` —cannot access this `protected` member of the class `Superclass1`.

However, the compiler reports an error at (7) in the method `printState1()` of the class `Subclass2` where the `protected` instance field `proStr` in an instance of the class `Superclass1` *cannot* be accessed by the reference `obj1`.

In contrast, the method `printState2()` at (8) in the class `Subclass2` uses the reference `obj2` that refers to an instance of the class `Subclass2` to access the instance fields declared in the class `Superclass1`, and now the `protected` instance field `proStr` in the superclass is accessible, as shown at (10).

This apparent anomaly is explained by the fact that a subclass in another package can only access `protected` instance members in the superclass via references of its own type—that is, `protected` instance members that are *inherited* by objects of the subclass. No inheritance from the superclass is involved in a subclass in another package when an object of the superclass is used.

Note that the above anomaly would *not* arise if a subclass in another package were to access any `protected static` members in the superclass, as such members are not part of any object of the superclass.

Members with Package Access

No access modifier implies package accessibility in this context. When no member access modifier is specified, the member is accessible only by other classes in the same package in which its class is declared. Even if its class is accessible in another package, the member is not accessible elsewhere. Package member access is more restrictive than `protected` member access.

In Example 6.8, the instance field `pkgBool` in an instance of the class `Superclass1` has package access and is only accessible within `pkg1`, but not in any other packages— that is to say, it is accessible only by Client 1 and Client 2. Client 3 and Client 4 in `pkg2` cannot access this field.

`private` *Members*

The `private` modifier is the most restrictive of all the access modifiers. Private members are not accessible by any other classes. This also applies to subclasses, whether they are in the same package or not. Since they are not accessible by their simple names in a subclass, they are also not inherited by the subclass. A standard design strategy for a class is to make all instance fields `private` and provide `public` get methods for such fields. Auxiliary methods are often declared as `private`, as they do not concern any client.

None of the clients in Figure 6.5 can access the `private` instance field `privLong` in an instance of the class `Superclass1`. This instance field is only accessible in the defining class—that is, in the class `Superclass1`.

Table 6.3 provides a summary of access modifiers for members in a class. References in parentheses refer to clients in Figure 6.5.

Table 6.3 *Accessibility of Members in a Class (Non-Modular)*

Members	In the defining class	In a subclass in the same package (Client 1)	In a non-subclass in the same package (Client 2)	In a subclass in a different package (Client 3)	In a non-subclass in a different package (Client 4)
`public`	Yes	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	Yes	No
`package`	Yes	Yes	Yes	No	No
`private`	Yes	No	No	No	No

Additional Remarks on Accessibility

Access modifiers that can be specified for a class member apply equally to *constructors*. However, when no constructor is specified, the default constructor inserted by the compiler implicitly follows the accessibility of the class.

Accessibility of members declared in an enum type is analogous to those declared in a class, except for enum constants that are always `public` and constructors that are always `private`. A `protected` member in an enum type is only accessible in its own package, since an enum type is implicitly `final`. Member declarations in an enum type are discussed in §5.13, p. 290.

In contrast, the accessibility of members declared in an interface is always implicitly `public` (§5.6, p. 238). Omission of the `public` access modifier in this context does *not* imply package accessibility.

6.6 Scope Rules

Java provides explicit access modifiers to control the accessibility of members in a class by external clients, but in two areas access is governed by specific scope rules:

- Class scope for members: how member declarations are accessed within the class
- Block scope for local variables: how local variable declarations are accessed within a block

Class Scope for Members

Class scope concerns accessing members (including inherited ones) from code within a class. Table 6.4 gives an overview of how static and non-static code in a class can access members of the class, including those that are inherited. Table 6.4 assumes the following declarations:

Click here to view code image

```
class SuperClass {
  int instanceVarInSuper;
  static int staticVarInSuper;

  void instanceMethodInSuper()      { /* ... */ }
  static void staticMethodInSuper() { /* ... */ }
  // ...
}

class MyClass extends SuperClass {
  int instanceVar;
  static int staticVar;

  void instanceMethod()      { /* ... */ }
  static void staticMethod() { /* ... */ }
  // ...
}
```

Table 6.4 *Accessing Members within a Class*

Member declarations in the class `MyClass`	Non-static code in the class `MyClass` can refer to the member as	Static code in the class `MyClass` can refer to the member as

Instance variables	`instanceVar`
	`this.instanceVar`
	`instanceVarInSuper`
	`this.instanceVarInSuper`
	`super.instanceVarInSuper`
		Not possible

Instance methods	`instanceMethod()`
	`this.instanceMethod()`
	`instanceMethodInSuper()`
	`this.instanceMethodInSuper()`
	`super.instanceMethodInSuper()`
		Not possible

Static variables	`staticVar` `MyClass.staticVar`	`staticVar` `MyClass.staticVar`

	`staticVarInSuper`
	`MyClass.staticVarInSuper`
	`SuperClass.staticVarInSuper`
		`staticVarInSuper`
		`this.staticVarInSuper`
		`super.staticVarInSuper`
		`MyClass.staticVarInSuper`
		`SuperClass.staticVarInSuper`

Static methods	`staticMethodInSuper()`
	`MyClass.staticMethodInSuper()`
	`SuperClass.staticMethodInSuper()`
		`staticMethodInSuper()`
		`this.staticMethodInSuper()`
		`super.staticMethodInSuper()`
		`MyClass.staticMethodInSuper()`
		`SuperClass.staticMethodInSuper()`

The golden rule is that static code can only access other static members by their simple names. Static code is not executed in the context of an object, so the references `this` and `super` are not available. An object has knowledge of its class, so static members are always accessible in a non-static context.

Note that using the class name to access static members within the class is no different from how external clients access these static members.

The following factors can all influence the scope of a member declaration:

- Shadowing of a field declaration, either by local variables (p. 354) or by declarations in the subclass (§5.1, p. 203)
- Overriding an instance method from a superclass (§5.1, p. 196)
- Hiding a static method declared in a superclass (§5.1, p. 203)

Within a class, references of the class can be declared and used to access *all* members in the class, regardless of their access modifiers. In Example 6.9, the method `duplicateLight` at (1) in the class `Light` has the parameter `oldLight` and the local variable `newLight` that are references of the class `Light`. Even though the fields of the class are `private`, they are accessible through the two references (`oldLight` and `newLight`) in the method `duplicateLight()`, as shown at (2), (3), and (4).

Example 6.9 *Class Scope*

Click here to view code image

```
class Light {
  // Instance variables:
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  public static Light duplicateLight(Light oldLight) {      // (1)
    Light newLight = new Light();
    newLight.noOfWatts = oldLight.noOfWatts;                // (2)
    newLight.indicator = oldLight.indicator;                // (3)
    newLight.location  = oldLight.location;                 // (4)
    return newLight;
  }
}
```

Block Scope for Local Variables

Declarations and statements can be grouped into a *block* using curly brackets, `{}`. Blocks can be nested, and scope rules apply to local variable declarations in such blocks. A local declaration can appear anywhere in a block. The general rule is that a variable declared in a block is *in scope* in the block in which it is declared, but it is not accessible outside this block. It is not possible to redeclare a variable if a local variable of the same name is already declared in the current scope.

Local variables of a method include the formal parameters of the method and variables that are declared in the method body. The local variables in a method are created each time the method is invoked, and are therefore distinct from local variables in other invocations of the same method that might be executing (§7.1, p. 365).

Figure 6.6 illustrates *block scope* (also known as *lexical scope*) for local variables. It shows four blocks: Block 1 is the body of the method `main()`, Block 2 is the body of the `for(;;)` loop, Block 3 is the body of a `switch` statement, and Block 4 is the body of an `if` statement.

- Parameters cannot be redeclared in the method body, as shown at (1) in Block 1.
- A local variable—already declared in an enclosing block, and therefore visible in a nested block—cannot be redeclared in the nested block. These cases are shown at (3), (5), and (6).

- A local variable in a block can be redeclared in another block if the blocks are *disjoint*—that is, they do not overlap. This is the case for variable `i` at (2) in Block 3 and at (4) in Block 4, as these two blocks are disjoint.

The scope of a local variable declaration begins from where it is declared in the block and ends where this block terminates. The scope of the loop variable `index` is the entire Block 2. Even though Block 2 is nested in Block 1, the declaration of the variable `index` at (7) in Block 1 is valid. The scope of the variable `index` at (7) spans from its declaration to the end of Block 1, and it does not overlap with that of the loop variable `index` in Block 2.

Figure 6.6 *Block Scope*

6.7 Implementing Immutability

Shared resources are typically implemented using `synchronized` code in order to guarantee thread safety of the shared resource (§22.4, p. 1387). However, if the shared resource is an immutable object, thread safety comes for free.

An object is *immutable* if its state cannot be changed once it has been constructed. Since its state can only be read, there can be no thread interference and the state is always consistent.

Some examples of immutable classes from the Java SE Platform API are listed in Table 6.5. Any method that seemingly modifies the state of an immutable object is in fact returning a new immutable object with the state modified appropriately based on the original object. Primitive values are of course always immutable.

Table 6.5 *Examples of Immutable Classes*

`java.lang.String`	This class implements immutable strings (§8.4, p. 439).
Wrapper classes in the `java.lang` package: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`	Wrapper classes create immutable objects that wrap a value of their respective primitive type (§8.3, p. 429).
`java.nio.file.Path`	Objects of classes that implement this interface are immutable and represent a system-dependent file path (§21.2, p. 1289).
Temporal classes in the `java.time` package: `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration`, `ZonedDateTime`	Temporal classes create immutable objects that represent date-/time-based values (Chapter 17, p. 1023).
`java.time.format.DateTimeFormatter`	This class creates immutable objects with customized formatting and parsing capabilities for date-/time-based values (§18.6, p. 1134).
`java.util.Locale`	Immutable objects of this class represent a specific geographical,

political, or cultural region (§18.1, p. 1096).

Example 6.10 *Implementing an Immutable Class*

Click here to view code image

```
import java.util.Arrays;

public final class WeeklyStats {              // (1) Class is final.

  private final String description;           // (2) Immutable string
  private final int weekNumber;               // (3) Immutable primitive value
  private final int[] stats;                  // (4) Mutable int array

  public WeeklyStats(String description, int weekNumber, int[] stats) {   // (
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber)
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                          Arrays.toString(stats));
    }
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length);   // Create a private cop
  }

  public int getWeekNumber() {                // (6) Returns immutable primiti
    return weekNumber;
  }

  public String getDescription() {            // (7) Returns immutable string.
    return description;
  }

  public int getDayStats(int dayNumber) {     // (8) Returns stats for given d
    return (0 <= dayNumber && dayNumber < 7) ? stats[dayNumber] : -1;
  }

  public int[] getStats() {                   // (9) Returns a copy of the sta
    return Arrays.copyOf(this.stats, this.stats.length);
  }

  @Override
  public String toString() {
    return description + "(week " + weekNumber + "):" + Arrays.toString(stats)
  }
}
```

Click here to view code image

```
public class StatsClient {
  public static void main(String[] args) {
    WeeklyStats ws1
        = new WeeklyStats("Appointments", 45, new int[] {5, 3, 8, 10, 7, 8, 9}
    System.out.println(ws1);
    WeeklyStats ws2
        = new WeeklyStats("E-mails", 47, new int[] {10, 5, 20, 7});
    System.out.println(ws2);
  }
}
```

Output from the program:

Click here to view code image

```
Appointments(week 45):[5, 3, 8, 10, 7, 8, 9]
Exception in thread "main" java.lang.IllegalArgumentException: Stats not for w
week: [10, 5, 20, 7]
     at WeeklyStats.<init>(WeeklyStats.java:14)
     at StatsClient.main(StatsClient.java:7)
```

There are certain guidelines that can help to avoid common pitfalls when implementing immutable classes. We will illustrate implementing an immutable class called `WeeklyStats` in Example 6.10, whose instances, once created, cannot be modified. The class `WeeklyStats` creates an object with weekly statistics of a specified entity.

- *It should not be possible to extend the class.*

  Caution should be exercised in extending an immutable class to prevent any subclass from subverting the immutable nature of the superclass.

  A straightforward approach is to declare the class as `final`, as was done in Example 6.10 at (1) for the class `WeeklyStats`. Another approach is to declare the constructor as `private` and provide static factory methods to construct instances (discussed below). A static factory method is a static method whose sole purpose is to construct and return a new instance of the class—an alternative to calling the constructor directly.

- *All fields should be declared* `final` *and* `private`.

  Declaring the fields as `private` makes them accessible only inside the class, and other clients cannot access and modify them. This is the case for the fields in the `WeeklyStats` class at (2), (3), and (4).

  Declaring a field as `final` means the value stored in the field cannot be changed once initialized. However, if the `final` field is a reference to an object, the state of this object can be changed by other clients who might be sharing this object, unless the object is also immutable. See the last guideline on how to safeguard the state of an mutable object referenced by a field.

- *Check the consistency of the object state at the time the object is created.*

  Since it is not possible to change the state of an immutable object, the state should be checked for consistency when the object is created. If all relevant information to initialize the object is available when it is created, the state can be checked for consistency and any necessary measures taken. For example, a suitable exception can be thrown to signal illegal arguments.

In the class `WeeklyStats`, the constructor at (5) is passed all the necessary values to initialize the object, and it checks whether they will result in a legal and consistent state for the object.

- *No set methods (a.k.a. setter or mutator methods) should be provided.*

  Set methods that change values in fields or objects referenced by fields should not be permitted. The class `WeeklyStats` does not have any set methods, and only provides get methods (a.k.a. *getter* or *assessor methods*).

  If a setter method is necessary, then the method should create a new instance of the class based on the modified state, and return that to the client, leaving the original instance unmodified. This approach has to be weighed against the cost of creating new instances, but is usually offset by other advantages associated with using immutable classes, like thread safety without synchronized code. Caching frequently used objects can alleviate some overhead of creating new objects, as exemplified by the immutable wrapper classes for primitive types. For example, the `Boolean` class has a static factory method `valueOf()` that always returns one of two objects, `Boolean.TRUE` or `Boolean.FALSE`, depending on whether its `boolean` argument was `true` or `false`, respectively. The `Integer` class interns values between –128 and 127 for efficiency so that there is only one `Integer` object to represent each `int` value in this range.

- *A client should not be able to access mutable objects referred to by any fields in the class.*

  The class should not provide any methods that can modify its mutable objects. The class `WeeklyStats` complies with this requirement.

  A class should also not share references to its mutable objects. The field at (4) has the type array of `int` that is mutable. An `int` array is passed as a parameter to the constructor at (5). The constructor in this case makes its own copy of this `int` array, so as not to share the array passed as an argument by the client. The `getWeeklyStats()` method at (8) does not return the reference value of the `int` array stored in the field `stats`. It creates and returns a new `int` array with values copied from its private `int` array. This technique is known as *defensive copying*. This way, the class avoids sharing references of its mutable objects with clients.

The class declaration below illustrates another approach to prevent a class from being extended. The class `WeeklyStats` is no longer declared `final` at (1), but now has a `private` constructor. This constructor at (5a) cannot be called by any client of the class to create an object. Instead, the class provides a static factory method at (5b) that creates an object by calling the `private` constructor. No subclass can be instantiated, as the superclass `private` constructor cannot be called, neither directly nor implicitly, in a subclass constructor.

Click here to view code image

```
public class WeeklyStatsV2 {                   // (1) Class is not final.
  ...
  private WeeklyStatsV2(String description,
      int weekNumber, int[] stats) {           // (5a) Private constructor
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length); // Create a private copy.
  }

  // (5b) Static factory method to construct objects.
  public static WeeklyStatsV2 getNewWeeklyStats(String description,
                                                int weekNumber, int[] stats) {
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber)
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                         Arrays.toString(stats));
    }
    return new WeeklyStatsV2(description, weekNumber, stats);
  }
  ...
}
```

A class having just static methods is referred to as a *utility class*. Such a class cannot be instantiated and has no state, and is thus immutable. Examples of such classes in the Java API include the following: the `java.lang.Math` class, the `java.util.Collections` class, the `java.util.Arrays` class, and the `java.util.concurrent.Executors` class.

Apart from being thread safe, immutable objects have many other advantages. Once created, their state is guaranteed to be consistent throughout their lifetime. That makes them easy to reason about. Immutable classes are relatively simple to construct, amenable to testing, and easy to use compared to mutable classes. There is hardly any need to make or provide provisions for making copies of such objects. Their hash code value, once computed, can be cached for later use, as it will never change. Because of their immutable state, they are ideal candidates for keys in maps, and as elements in sets. They also are ideal building blocks for new and more complex objects.

A downside of using an immutable object is that if a value must be changed in its state, then a new object must be created, which can be costly if object construction is expensive.

Review Questions

6.9 Which of the following statements are true about the use of modifiers? Select the two correct answers.

- **a.** If no access modifier (`public`, `protected`, or `private`) is specified for a member declaration, the member is accessible only by classes in the package of its class and by subclasses of its class in any package.
- **b.** You cannot specify accessibility of local variables. They are not accessible outside the block in which they are declared.
- **c.** Subclasses of a class must reside in the same package as the class they extend.
- **d.** Local variables can be declared as `static`.
- **e.** The objects themselves do not have any access modifiers; only field references do.

6.10 Given the following declaration of a class, which field is accessible from outside the package `com.corporation.project`?

Click here to view code image

```
package com.corporation.project;
public class MyClass {
            int i;
  public    int j;
  protected int k;
  private   int l;
}
```

Select the one correct answer.

a. Field `i` is accessible in all classes in other packages.
b. Field `j` is accessible in all classes in other packages.
c. Field `k` is accessible in all classes in other packages.
d. Field `k` is accessible in subclasses only in other packages.
e. Field `l` is accessible in all classes in other packages.
f. Field `l` is accessible in subclasses only in other packages.

6.11 Which statement is true about the accessibility of members?

Select the one correct answer.

a. A private member is always accessible within the same package.
b. A private member can be accessed only in the class in which it is declared.
c. A member with package access can be accessed by any subclass of the class in which it is declared.
d. A private member cannot be accessed at all.
e. Package accessibility for a member can be declared using the keyword `default`.

6.12 Which statement is true about immutability?

Select the one correct answer.

a. Instances of a `final` class are immutable.
b. Instances of a `static` class are immutable.
c. All members of an immutable class are also immutable.
d. None of the above

6.13 Which code modifications would make class `Dog` immutable?

Click here to view code image

```
public class Dog {
  String name;
  public Dog(String name) {
    this.name = name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}
```

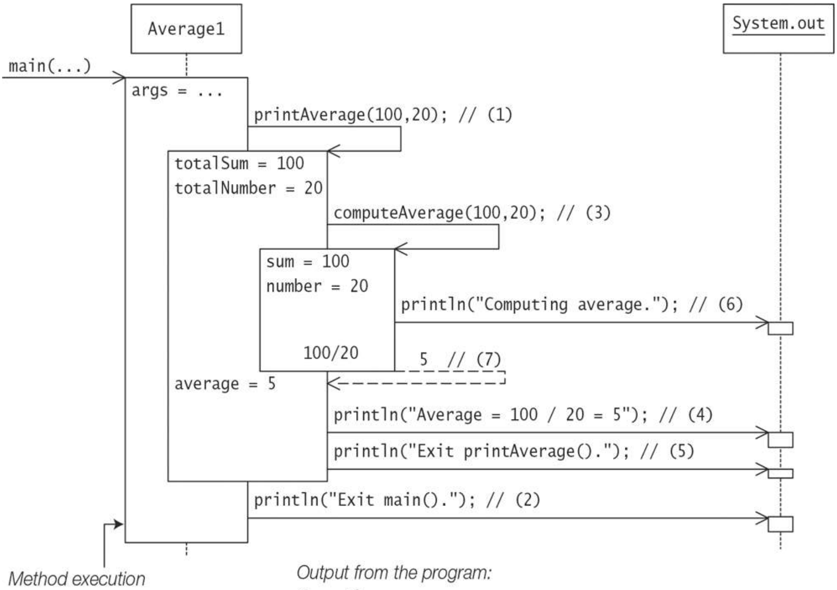
Select the one correct answer.

a. Mark the field `name` as `private` and remove the `setName()` method.

b. Mark the field `name` as `final` and make the `setName()` method `private`.

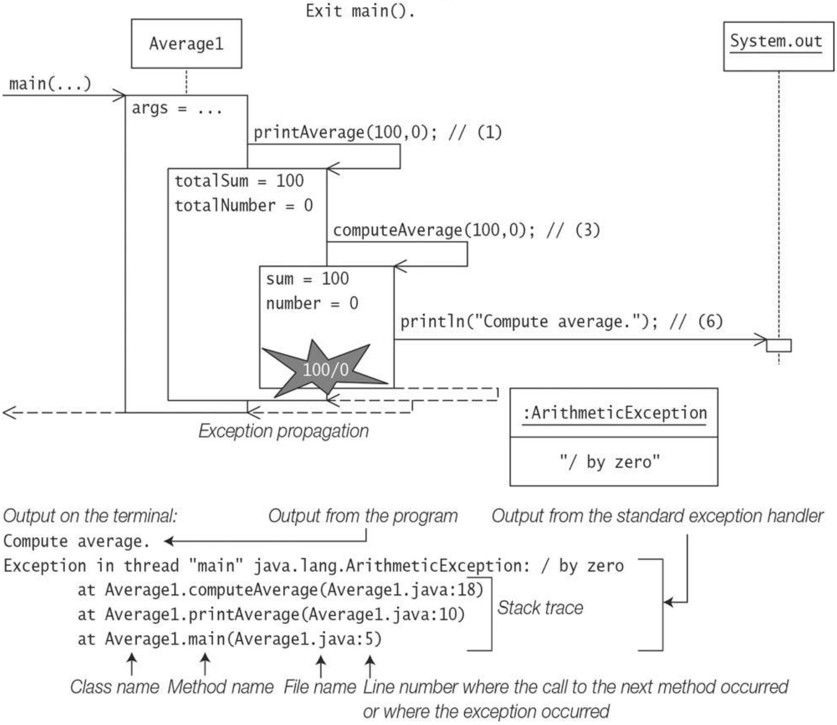
c. Mark the field `name` as `final` and initialize it in the declaration, then remove the `setName()` method.

d. None of the above

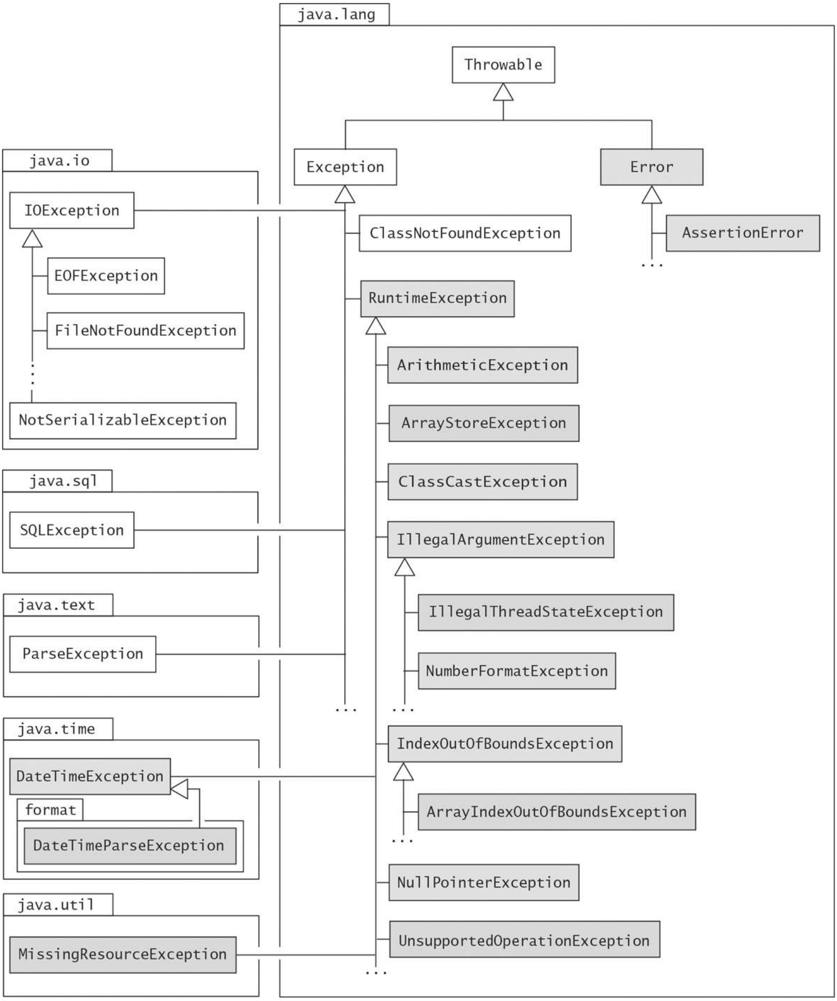

Average1
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
computeAverage(100,20); // (3)
sum = 100
number = 20
println("Computing average."); // (6)
100/20
5 // (7)
average = 5
println("Average = 100 / 20 = 5"); // (4)
println("Exit printAverage()."); // (5)
println("Exit main()."); // (2)
Method execution

Output from the program:

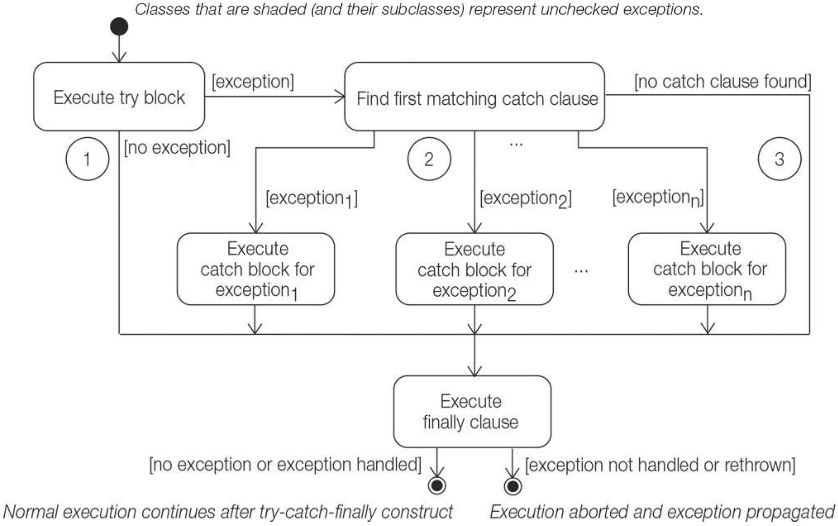
```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

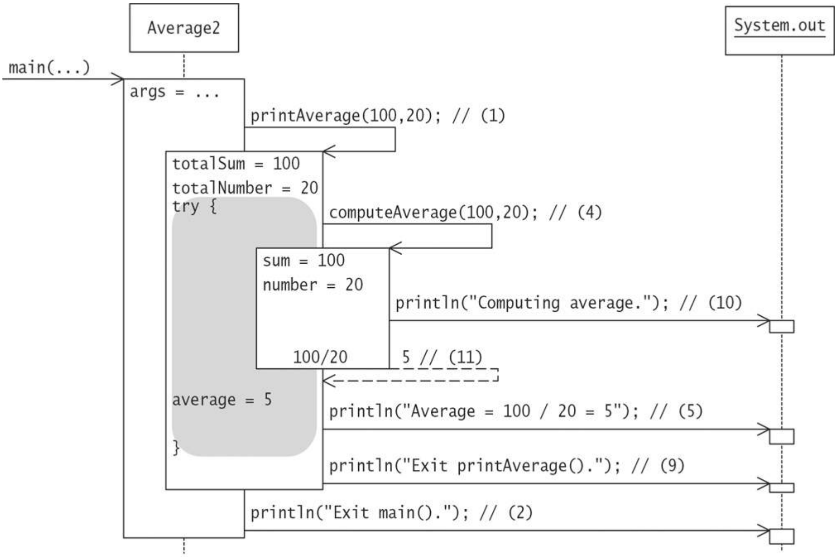

Average1
System.out
main(...)
args = ...
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
computeAverage(100,0); // (3)
sum = 100
number = 0
println("Compute average."); // (6)
100/0
:ArithmeticException
"/ by zero"
Exception propagation
Output on the terminal:
Output from the program
Output from the standard exception handler
Stack trace
Class name
Method name
File name
Line number where the call to the next method occurred or where the exception occurred

```
Compute average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```


java.lang
Throwable
Exception
Error
AssertionError
ClassNotFoundException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
NullPointerException
UnsupportedOperationException
java.io
IOException
EOFException
FileNotFoundException
NotSerializableException
java.sql
SQLException
java.text
ParseException
java.time
DateTimeException
format
DateTimeParseException
java.util
MissingResourceException

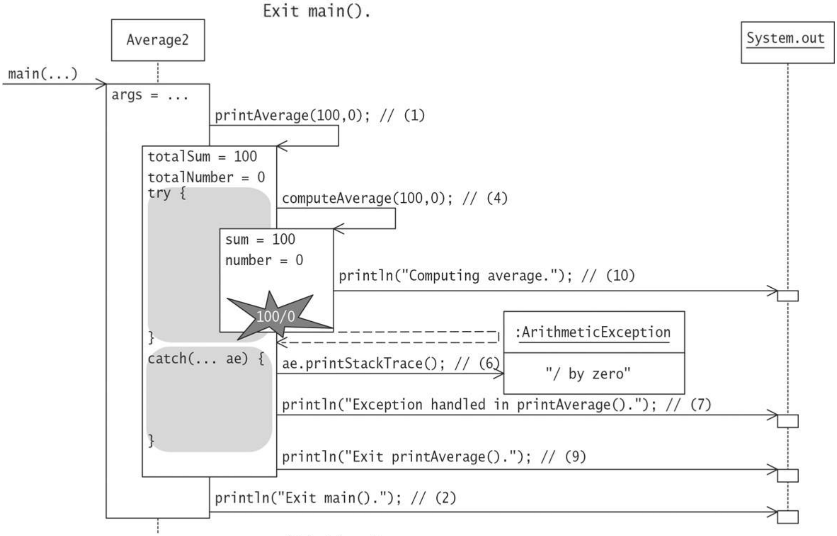
Classes that are shaded (and their subclasses) represent unchecked exceptions.


Execute try block
[exception]
Find first matching catch clause
[no catch clause found]
[no exception]
1
2
3
[exception1]
[exception2]
[exceptionn]
Execute catch block for exception1
Execute catch block for exception2
Execute catch block for exceptionn
Execute finally clause
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct
Execution aborted and exception propagated


Average2
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
try {
computeAverage(100,20); // (4)
sum = 100
number = 20
println("Computing average."); // (10)
100/20
5 // (11)
average = 5
println("Average = 100 / 20 = 5"); // (5)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

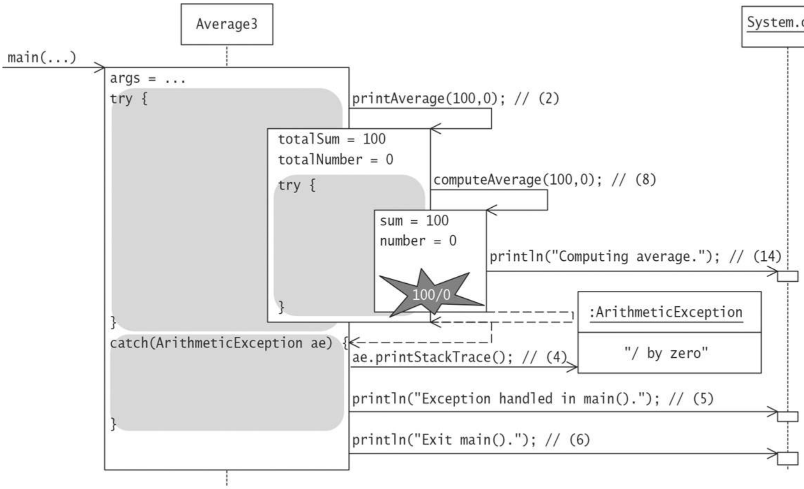
Output from the program:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```


Average2
System.out
main(...)
args = ...
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (4)
sum = 100
number = 0
println("Computing average."); // (10)
100/0
:ArithmeticException
"/ by zero"
}
catch(... ae) {
ae.printStackTrace(); // (6)
println("Exception handled in printAverage()."); // (7)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:23)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```


Average3
System.out
main(...)
args = ...
try {
printAverage(100,0); // (2)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (8)
sum = 100
number = 0
println("Computing average."); // (14)
100/0
}
:ArithmeticException
"/ by zero"
}
catch(ArithmeticException ae) {
ae.printStackTrace(); // (4)
println("Exception handled in main()."); // (5)
}
println("Exit main()."); // (6)

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:28)
        at Average3.printAverage(Average3.java:16)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

Exception Handling 7

Chapter Topics

- Understanding program execution and the exception handling facility
- Recognizing the inheritance hierarchy of the `Throwable` class and its subclasses `Exception`, `RuntimeException`, and `Error` that define exception classes in Java
- Distinguishing between checked and unchecked exceptions

- Defining customized exception types
- Understanding the `try-catch-finally` construct and the control flow paths through it
- Using multiple `catch` clauses with the `try` statement
- Creating and throwing exceptions programmatically with the `throw` statement
- Using the `throws` clause to specify checked exceptions in the method header to propagate exceptions
- Using uni-`catch` and multi-`catch` clauses with the `try` statement to catch exceptions
- Rethrowing exceptions and handling chained exceptions
- Using the `try`-with-resources statement and implementing the `AutoCloseable` interface
- Handling suppressed exceptions

Java SE 17 Developer Exam O	
[4.1] Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions	*§7.2, p. 375* *§7.3, p. 375* *§7.6, p. 397* *§7.7, p. 407*
Java SE 11 Developer Exam O	
[4.1] Handle exceptions in the Java program by using try/catch/finally clauses, try-with-resource, and multi-catch statements	*§7.3, p. 375* *§7.6, p. 397* *§7.7, p. 407*
[4.2] Create and use custom exceptions	*§7.2, p. 375*

An exception in Java signals the occurrence of an event that disrupts the normal flow of program execution. Such an event typically occurs due to violation of some semantic constraint of the Java programming language during execution—for example, a requested file cannot be found, an array index is out of bounds, or a network link failed. Inserting explicit checks in the code for such events can easily result in less comprehensible code. Java provides an exception handling mechanism for systematically dealing with such events.

7.1 Stack-Based Execution and Exception Propagation

The exception mechanism is built around the *throw-and-catch* paradigm. To *throw* an exception is to signal that an unexpected event has occurred. To *catch* an exception is to take appropriate action to deal with the exception. An exception is caught by an *exception handler*, and the exception need not be caught in the same context in which it was thrown. The runtime behavior of the program determines which exceptions are thrown and how they are caught. The throw-and-catch principle is embedded in the `try-catch-finally` construct (p. 375).

Several threads can be executing at the same time in the JVM (§22.2, p. 1369). Each thread has its own *JVM stack* (also called a *runtime stack*, *call stack*, or *invocation stack* in the literature) that is used to handle execution of methods. Each element on the stack is called an *activation frame* or a *stack frame* and corresponds to a method call. Each new method call results in a new activation frame being pushed on the stack, which stores all the pertinent information such as the local variables. The method with the activation frame on the top of the stack is the one currently executing. When this method finishes executing, its activation frame is popped from the top of the stack. Execution then continues in the method corresponding to the activation frame that is now uncovered on the top of the stack. The methods on the stack are said to be *active*, as their execution has not completed. At any given time, the active methods on a JVM stack make up what is called the *stack trace* of a thread's execution.

Example 7.1 is a simple program to illustrate method execution. It calculates the average for a list of integers, given the sum of all the integers and the number of integers. It uses three methods:

- The method `main()` calls the method `printAverage()` with parameters supplying the total sum of the integers and the total number of integers, (1).
- The method `printAverage()` in turn calls the method `computeAverage()`, (3).
- The method `computeAverage()` uses integer division to calculate the average and returns the result, (7).

Example 7.1 *Method Execution*

Click here to view code image

```
public class Average1 {
    public static void main(String[] args) {
      printAverage(100, 20);                                         // (1)

      System.out.println("Exit main().");                           // (2)
    }

    public static void printAverage(int totalSum, int totalCount) {
      int average = computeAverage(totalSum, totalCount);            // (3)
      System.out.println("Average = " +                              // (4)
          totalSum + " / " + totalCount + " = " + average);
      System.out.println("Exit printAverage().");                    // (5)
    }

    public static int computeAverage(int sum, int count) {
      System.out.println("Computing average.");                     // (6)
      return sum/count;                                              // (7)
    }
  }
```

Output of program execution:

Click here to view code image

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Execution of Example 7.1 is illustrated in Figure 7.1. Each method execution is shown as a box with the local variables declared in the method. The height of the box indicates how long a method is active. Before the call to the method `System.out.println()` at (6) in Figure 7.1, the stack trace comprises the three active methods: `main()`, `printAverage()`, and `computeAverage()`. The result `5` from the method `computeAverage()` is returned at (7) in Figure 7.1. The output from the program corresponds with the sequence of method calls in Figure 7.1. As the program terminates normally, this program behavior is called *normal execution*.

If the method call at (1) in Example 7.1

Click here to view code image

```
printAverage(100, 20);                                 // (1)
```

is replaced with

Click here to view code image

```
printAverage(100, 0);                                  // (1)
```

and the program is run again, the output is as follows:

Click here to view code image

```
Computing average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

Figure 7.2 illustrates the program execution when the method `printAverage()` is called with the arguments `100` and `0` at (1). All goes well until the `return` statement at (7) in the method `computeAverage()` is executed. An error event occurs in calculating the expression `sum/number` because integer division by 0 is an illegal operation. This event is signaled by the JVM by *throwing* an `ArithmeticException` (p. 372). This exception is *propagated* by the JVM through the JVM stack as explained next.

Figure 7.1 *Normal Method Execution*

Figure 7.2 illustrates the case where an exception is thrown and the program does not take any explicit action to deal with the exception. In Figure 7.2, execution of the `computeAverage()` method is suspended at the point where the exception is thrown. The execution of the `return` statement at (7) never gets completed. Since this method does not have any code to deal with the exception, its execution is likewise terminated abruptly and its activation frame popped. We say that the method *completes abruptly*. The exception is then offered to the method whose activation is now on the top of the stack (`printAverage()`). This method does not have any code to deal with the exception either, so its execution completes abruptly. The statements at (4) and (5) in the method `printAverage()` never get executed. The exception now propagates to the last active method (`main()`). This does not deal with the exception either. The `main()` method also completes abruptly. The statement at (2) in the `main()` method never gets executed. Since the exception is not *caught* by any of the active methods, it is dealt with by the main thread's *default exception handler*. The default exception handler usually prints the name of the exception, with an explanatory message, followed by a printout of the stack trace at the time the exception was thrown. An uncaught exception, as in this case, results in the death of the thread in which the exception occurred.

Figure 7.2 *Exception Propagation*

If an exception is thrown during the evaluation of the left-hand operand of a binary expression, then the right-hand operand is not evaluated. Similarly, if an exception is thrown during the evaluation of a list of expressions (e.g., a list of actual parameters in a method call), evaluation of the rest of the list is skipped.

If the line numbers in the stack trace are not printed in the output as shown previously, use the following command to run the program:

Click here to view code image

```
>java -Djava.compiler=NONE Average1
```

7.2 Exception Types

Exceptions in Java are objects. All exceptions are derived from the `java.lang.Throwable` class. Figure 7.3 shows a partial hierarchy of classes derived from the `Throwable` class. The two main subclasses `Exception` and `Error` constitute the main categories of *throwables*, the term used to refer to both exceptions and errors. Figure 7.3 also shows that not all exception classes are found in the `java.lang` package.

Figure 7.3 *Partial Exception Inheritance Hierarchy*

All throwable classes in the Java SE Platform API at least define a zero-argument constructor and a one-argument constructor that takes a `String` parameter. This parameter can be set to provide a *detail message* when an exception is constructed. The purpose of the detail message is to provide more information about the actual exception.

> ```
> Throwable()
> Throwable(String msg)
> ```
>
> The first constructor constructs a throwable that has `null` as its detail message. The second constructor constructs a throwable that sets the specified string as its detail message.
>
> Most exception types provide analogous constructors.

The class `Throwable` provides the following common methods to query an exception:

> ```
> String getMessage()
> ```
>
> Returns the detail message.
>
> ```
> void printStackTrace()
> ```
>
> Prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the JVM

> stack when the exception was thrown. The stack trace can also be written to a `PrintStream` or a `PrintWriter` by supplying such a destination as an argument to one of the two overloaded `printStackTrace()` methods. Any *suppressed exceptions* associated with an exception on the stack trace are also printed (p. 415). It will also print the *cause* of an exception (which is also an exception) if one is available (p. 405).
>
> `String toString()`
>
> Returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the `getMessage()` method.

In dealing with throwables, it is important to recognize *situations* in which a particular throwable can occur, and the *source* that is responsible for throwing it. By *source* we mean:

- The *JVM* that is responsible for throwing the throwable, or
- The throwable that is explicitly thrown *programmatically* by the code in the application or by any API used by the application.

In further discussion of exception types, we provide an overview of situations in which selected throwables can occur and the source responsible for throwing them.

The `java.lang.Exception` Class

The class `Exception` represents exceptions that a program would normally want to catch. Its subclass `java.lang.RuntimeException` represents many common programming errors that can manifest at runtime (see the next subsection). Other subclasses of the `Exception` class, excluding the `RuntimeException` class, define what are known as *checked exceptions* (p. 374) that particularly aid in building robust programs. Some common checked exceptions are presented below.

Click here to view code image

`java.lang.ClassNotFoundException`

The class `ClassNotFoundException` is a subclass of the `Exception` class that signals that the JVM tried to load a class by its string name, but the class could not be found. A typical example of this situation is when the class name is misspelled while starting program execution with the `java` command. The source in this case is the JVM throwing the exception to signal that the class cannot be found.

java.io.IOException

The class `IOException` is a subclass of the `Exception` class that represents I/O-related exceptions that are found in the `java.io` package (`EOFException`, `FileNotFound-Exception`, `NotSerializableException`). Chapter 20, p. 1231, and Chapter 21, p. 1285, provide ample examples of contexts in which I/O-related exceptions can occur.

java.io.EOFException

The class `EOFException` is a subclass of the `IOException` class that represents an exception that signals that an *end of file* (EOF) or end of stream was reached unexpectedly when more input was expected—that is, there is no more input available. Typically, this situation occurs when an attempt is made to read input from a file when all data from the file has already been read, often referred to as *reading past EOF*.

Click here to view code image

java.io.FileNotFoundException

The class `FileNotFoundException` is a subclass of the `IOException` class that represents an exception that signals an attempt to open a file by using a specific pathname failed—in other words, the file with the specified pathname does not exist. This exception can also occur when an I/O operation does not have the required permissions for accessing the file.

Click here to view code image

java.io.NotSerializableException

The class `NotSerializableException` is a subclass of the `IOException` class that represents an exception that signals that an object does not implement the `Serializable` interface that is required in order for the object to be serialized (§20.5, p. 1261).

java.sql.SQLException

The class `SQLException` is a subclass of the `Exception` class that represents an exception that can provide information about various database-related errors that can occur. Chapter 24 provides examples illustrating such situations.

`java.text.ParseException`

The class `ParseException` is a subclass of the `Exception` class that represents an exception that signals unexpected errors while parsing. Examples of parsing date, number, and currency where this exception is thrown can be found in §18.5, p. 1116.

The `java.lang.RuntimeException` Class

Runtime exceptions are all subclasses of the `java.lang.RuntimeException` class, which is a subclass of the `Exception` class. As these runtime exceptions are usually caused by program bugs that should not occur in the first place, it is usually more appropriate to treat them as faults in the program design and let them be handled by the default exception handler.

Click here to view code image

`java.lang.ArithmeticException`

This exception represents situations where an illegal arithmetic operation is attempted, such as integer division by 0. It is typically thrown by the JVM.

Click here to view code image

`java.lang.ArrayIndexOutOfBoundsException`

Java provides runtime checking of the array index value, meaning out-of-bounds array indices. The subclass `ArrayIndexOutOfBoundsException` of the `RuntimeException` class represents exceptions thrown by the JVM that signal out-of-bound errors specifically for arrays—that is, an error in which an invalid index is used to access an element in the array. The index value must satisfy the relation 0 ≤ *index value* < *length of the array* (§3.9, p. 120).

Click here to view code image

`java.lang.ArrayStoreException`

This exception is thrown by the JVM when an attempt is made to store an object of the wrong type into an array of objects. The array store check at runtime ensures that an object being stored in the array is assignment compatible with the element type of the array (§5.7, p. 260). To make the array store check feasible at runtime, the array retains information about its declared element type at runtime.

`java.lang.ClassCastException`

This exception is thrown by the JVM to signal that an attempt was made to cast a reference value to a type that was not legal, such as casting the reference value of an `Integer` object to the `Long` type (§5.11, p. 269).

Click here to view code image

```
java.lang.IllegalArgumentException
```

The class `IllegalArgumentException` represents exceptions thrown to signal that a method was called with an illegal or inappropriate argument. For example, the `ofPattern(String pattern)` method in the `java.time.format.DateTimeFormatter` class throws an `IllegalArgumentException` when the letter pattern passed as an argument is invalid (§18.6, p. 1134).

Click here to view code image

`java.lang.IllegalThreadStateException`

The class `IllegalThreadStateException` is a subclass of the `IllegalArgumentException` class. Certain operations on a thread can only be executed when the thread is in an appropriate state (§22.4, p. 1380). This exception is thrown when this is not the case. For example, the `start()` method of a `Thread` object throws this exception if the thread has already been started (§22.3, p. 1370).

Click here to view code image

`java.lang.NumberFormatException`

The class `NumberFormatException` is a subclass of the `IllegalArgumentException` class that is specialized to signal problems when converting a string to a numeric value if the format of the characters in the string is not appropriate for the conversion. This exception is thrown programmatically. The numeric wrapper classes all have methods that throw this exception when conversion from a string to a numeric value is not possible (§8.3, p. 434).

Click here to view code image

```
java.lang.NullPointerException
```

This exception is typically thrown by the JVM when an attempt is made to use the `null` value as a reference value to refer to an object. This might involve calling an instance method using a reference that has the `null` value, or accessing a field using a reference that has the `null` value.

This programming error has made this exception one of the most frequently thrown exceptions by the JVM. The error message issued provides helpful information as to *where* and *which* reference raised the exception. However, inclusion of variables names in the message can be a potential security risk.

Click here to view code image

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.toLowerCase()" because "msg" is null
    at StringMethods.main(StringMethods.java:162)
```

Click here to view code image

java.lang.UnsupportedOperationException

This exception is thrown programmatically to indicate that an operation invoked on an object is not supported. Typically, a class implements an interface, but chooses not to provide certain operations specified in the interface. Methods in the class corresponding to these operations throw this exception to indicate that an operation is not supported by the objects of the class.

The API documentation of the `java.util.Collection` interface (§15.1, p. 783) in the Java Collections Framework states that certain methods are *optional*, meaning that a concrete collection class need not provide support for such operations, and if any such operation is not supported, then the method should throw this exception.

java.time.DateTimeException

In the Date and Time API, the class `DateTimeException` represents exceptions that signal problems with creating, querying, and manipulating date-time objects (§17.2, p. 1027).

Click here to view code image

`java.time.format.DateTimeParseException`

In the Date and Time API, the class `DateTimeParseException` is a subclass of the `Date-TimeException` class that represents an exception that signals unexpected errors when parsing date and time values (§18.6, p. 1127).

Click here to view code image

`java.util.MissingResourceException`

This exception is thrown programmatically, typically by the lookup methods of the `java.util.ResourceBundle` class (§18.3, p. 1104). Get methods on resource bundles typically throw this exception when no resource can be found for a given key. Static lookup methods in this class also throw this exception when no resource bundle can be found based on a specified base name for a resource bundle. Resource bundles are discussed in §18.3, p. 1102.

The `java.lang.Error` Class

The class `Error` and its subclasses define errors that are invariably never explicitly caught and are usually irrecoverable. Not surprisingly, most such errors are signaled by the JVM. Apart from the subclass mentioned below, other subclasses of the `java.lang.Error` class define different categories of errors.

The subclass `VirtualMachineError` represents virtual machine errors like stack overflow (`StackOverflowError`) and out of memory for object allocation (`OutOfMemoryError`). The subclass `LinkageError` represents class linkage errors like missing class definitions (`NoClassDefFoundError`). The subclass `AssertionError` of the `Error` class is used by the Java assertion facility.

Checked and Unchecked Exceptions

Except for `RuntimeException`, `Error`, and their subclasses, all exceptions are *checked* exceptions. A checked exception represents an unexpected event that is not under the control of the program—for example, a file was not found. The compiler ensures that if a method can throw a checked exception, directly or indirectly, the method must either catch the exception and take the appropriate action, or pass the exception on to its caller (p. 388).

Exceptions defined by the `Error` and `RuntimeException` classes and their subclasses are known as *unchecked* exceptions, meaning that a method is not obliged to deal with these kinds of exceptions (shown with gray color in Figure 7.3). Either they are irrecoverable (exemplified by the `Error` class), in which case the program should not attempt to deal with them, or they are programming errors (exemplified by the `RuntimeException` class and its subclasses) and should usually be dealt with as such, and not as exceptions.

Defining Customized Exceptions

Customized exceptions are usually defined to provide fine-grained categorization of error situations, instead of using existing exception classes with descriptive detail messages to differentiate among the various situations. New customized exceptions are usually defined by either extending the `Exception` class or one of its checked subclasses, thereby making the new exceptions checked, or extending the `RuntimeException` subclass or one of its subclasses to create new unchecked exceptions.

Customized exceptions, as any other Java classes, can declare fields, constructors, and methods, thereby providing more information as to their cause and remedy when they are thrown and caught. The `super()` call can be used in a constructor to set pertinent details about the exception: a detail message or the cause of the exception (p. 405), or both. Note that the exception class must be instantiated to create an exception object that can be thrown and subsequently caught and dealt with. The following code sketches a class declaration for an exception that can include all pertinent information about the exception. Typically, the new exception class provides a constructor to set the detail message.

Click here to view code image

```
public class EvacuateException extends Exception {
  // Fields
  private Date date;
  private Zone zone;
  private TransportMode transport;

  // Constructor
  public EvacuateException(Date d, Zone z, TransportMode t) {
    // Call the constructor of the superclass, usually passing a detail messag
    super("Evacuation of zone " + z);
    // ...
  }
  // Methods
  // ...
}
```

Several examples in subsequent sections illustrate exception handling.

7.3 Exception Handling: `try`, `catch`, and `finally`

The mechanism for handling exceptions is embedded in the `try-catch-finally` construct, which has the following basic form:

Click here to view code image

```
try {                                      // try block
   statements
} catch (exception_type1 parameter1) {      // uni-catch clause
   statements
}
...
   catch (exception_typen parametern) {      // uni-catch clause
   statements

} finally {                                // finally clause
   statements
}
```

A few aspects about the syntax of this construct should be noted. For each `try` block, there can be zero or more `catch` clauses (i.e., it can have *multiple* `catch` clauses), but only one `finally` clause. The `catch` clauses and the `finally` clause must always appear in conjunction with a `try` block, and in the right order. A `try` block must be followed by at least one `catch` clause, or a `finally` clause must be specified—in contrast to the `try`-with-resources statement where neither a `catch` nor a `finally` clause is mandatory (p. 407). In addition to the `try` block, each `catch` clause and the `finally` clause specify a block, `{ }`. The block notation is mandatory.

Exceptions thrown during execution of the `try` block can be caught and handled in a `catch` clause. Each `catch` clause defines an exception handler. The header of the `catch` clause specifies exactly one exception parameter. The exception type must be of the `Throwable` class or one of its subclasses; otherwise, the code will not compile. The type of the exception parameter of a `catch` clause is specified by a *single* exception type in the syntax shown earlier, and such a `catch` clause is called a *uni*-`catch` clause.

A `finally` clause is always executed, regardless of the cause of exit from the `try` block, or whether any `catch` clause was executed at all. The two exceptions to this scenario are if the JVM crashes or the `System.exit()` method is called. Figure 7.4 shows three typical scenarios of control flow through the `try-catch-finally` construct.

Figure 7.4 *The* `try-catch-finally` *Construct*

The `try` block, the `catch` clause, and the `finally` clause of a `try-catch-finally` construct can contain arbitrary code, which means that a `try-catch-finally` construct can be nested in any block of the `try-catch-finally` construct. However, such nesting can easily make the code difficult to read and is best avoided, if possible.

The `try` Block

The `try` block establishes a context for exception handling. Termination of a `try` block occurs as a result of encountering an exception, or from successful execution of the code in the `try` block.

The `catch` clauses are skipped for all normal exits from the `try` block when no exceptions are thrown, and control is transferred to the `finally` clause if one is specified (Scenario 1 in Figure 7.4).

For all exits from the `try` block resulting from exceptions, control is transferred to the `catch` clauses—if any such clauses are specified—to find a matching `catch` clause (Scenario 2 in Figure 7.4). In the case when no `catch` clause is specified, control is transferred to the mandatory `finally` clause. If no `catch` clause matches the thrown exception, control is transferred to the `finally` clause if one is specified (Scenario 3 in Figure 7.4).

The `catch` Clause

Only an exit from a `try` block resulting from an exception can transfer control to a `catch` clause. A `catch` clause can catch the thrown exception only if the exception is assignable to the parameter in the `catch` clause. The code of the first such `catch` clause is executed, and all other `catch` clauses are ignored.

On exit from a `catch` clause, normal execution continues unless there is any uncaught exception that has been thrown and not handled. If this is the case, the method is aborted after the execution of any `finally` clause and the exception propagated up the JVM stack.

It is important to note that after a `catch` clause has been executed, control is always transferred to the `finally` clause if one is specified. This is true as long as there is a `finally` clause, regardless of whether the `catch` clause itself throws an exception.

In Example 7.2, the method `printAverage()` calls the method `computeAverage()` in a `try-catch` construct at (4). The `catch` clause is declared to catch exceptions of type `ArithmeticException`. The `catch` clause handles the exception by printing the stack trace and some additional information at (7) and (8), respectively. Normal execution of the program is illustrated in Figure 7.5, which shows that the `try` block is executed but no exceptions are thrown, with normal execution continuing after the `try-catch` construct. This corresponds to Scenario 1 in Figure 7.4.

Figure 7.5 *Exception Handling (Scenario 1)*

Example 7.2 *The* `try-catch` *Construct*

Click here to view code image

```
public class Average2 {

  public static void main(String[] args) {
    printAverage(100, 20);                                         // (1)
    System.out.println("Exit main().");                            // (2)
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                          // (3)
      int average = computeAverage(totalSum, totalCount);          // (4)
      System.out.println("Average = " +                            // (5)
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                             // (6)
      ae.printStackTrace();                                        // (7)
      System.out.println("Exception handled in printAverage().");  // (8)
    }
    System.out.println("Exit printAverage().");                    // (9)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                      // (10)
    return sum/count;                                              // (11)
  }
}
```

Output from the program, with call `printAverage(100, 20)` at (1):

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Output from the program, with call `printAverage(100, 0)` at (1):

Click here to view code image

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:23)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

However, if we run the program in Example 7.2 with the following call at (1):

```
printAverage(100, 0)
```

an `ArithmeticException` is thrown by the integer division operator in the method `computeAverage()`. In Figure 7.6 we see that the execution of the method `compute-Average()` is stopped and the exception propagated to method `printAverage()`, where it is handled by the `catch` clause at (6). Normal execution of the method continues at (9) after the `try-catch` construct, as witnessed by the output from the statements at (9) and (2). This corresponds to Scenario 2 in Figure 7.4.

In Example 7.3, the `main()` method calls the `printAverage()` method in a `try-catch` construct at (1). The `catch` clause at (3) is declared to catch exceptions of type `ArithmeticException`. The `printAverage()` method calls the `computeAverage()` method in a `try-catch` construct at (7), but here the `catch` clause is declared to catch exceptions of type `IllegalArgumentException`. Execution of the program is illustrated in Figure 7.7, which shows that the `ArithmeticException` is first propagated to the `catch` clause in the `printAverage()` method. Because this `catch` clause cannot handle this exception, it is propagated further to the `catch` clause in the `main()` method, where it is caught and handled. Normal execution continues at (6) after the exception is handled.

In Example 7.3, the execution of the try block at (7) in the printAverage() method is never completed: The statement at (9) is never executed. The `catch` clause at (10) is skipped. The execution of the `printAverage()` method is aborted: The statement at (13) is never executed, and the exception is propagated. This corresponds to Scenario 3 in Figure 7.4.

Figure 7.6 *Exception Handling (Scenario 2)*

Example 7.3 *Exception Propagation*

Click here to view code image

```
public class Average3 {

  public static void main(String[] args) {
    try {                                                          // (1)
      printAverage(100, 0);                                        // (2)
    } catch (ArithmeticException ae) {                             // (3)
      ae.printStackTrace();                                        // (4)
      System.out.println("Exception handled in main().");          // (5)
    }
    System.out.println("Exit main().");                            // (6)
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                          // (7)
      int average = computeAverage(totalSum, totalCount);          // (8)
      System.out.println("Average = " +                            // (9)

          totalSum + " / " + totalCount + " = " + average);
    } catch (IllegalArgumentException iae) {                       // (10)
      iae.printStackTrace();                                       // (11)
      System.out.println("Exception handled in printAverage().");  // (12)
    }
    System.out.println("Exit printAverage().");                    // (13)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                      // (14)
    return sum/count;                                              // (15)
  }
}
```

Output from the program:

Click here to view code image

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:28)
        at Average3.printAverage(Average3.java:16)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

The scope of the exception parameter name in the `catch` clause is the body of the `catch` clause—that is, it is a local variable in the body of the `catch` clause. As mentioned earlier, the type of the exception object must be *assignable* to the type of the argument in the `catch` clause. In the body of the `catch` clause, the exception object can be queried like any other object by using the parameter name.

The `javac` compiler complains if a `catch` clause for a superclass exception shadows the `catch` clause for a subclass exception, as the `catch` clause of the subclass exception will never be executed (a situation known as *unreachable code*). The following example shows incorrect order of the `catch` clauses at (1) and (2), which will result in a compile-time error at (2): The superclass `Exception` will shadow the subclass `ArithmeticException`.

Click here to view code image

```
try {
  // ...
} catch (Exception e) {              // (1) Superclass shadows subclass
  System.out.println(e);
} catch (ArithmeticException e) {    // (2) Compile-time error: Unreachable co
  System.out.println(e);
}
```

The compiler will also flag an error if the parameter of the `catch` clause has a *checked* exception type that cannot be thrown by the `try` block, as this would result in unreachable code.

Click here to view code image

```
try {
  throw new ArithmeticException();      // IOException never thrown in try blo
} catch (IOException e) {               // Compile-time error: Unreachable cod
  System.out.println(e);
}
```

Figure 7.7 *Exception Handling (Scenario 3)*

The `finally` Clause

If the `try` block executes, then the `finally` clause is guaranteed to be executed, regardless of whether any `catch` clause was executed, barring the two special cases (JVM crashes or the `System.exit()` method is called). Since the `finally` clause is always executed before control transfers to its final destination, the `finally` clause can be used to specify any clean-up code (e.g., to free resources such as files and network connections). However, the `try`-with-resources statement provides a better solution for handling resources, and eliminates the use of the `finally` clause in many cases (p. 407).

A `try-finally` construct can be used to control the interplay between two actions that must be executed in the correct order, possibly with other intervening actions. In the code below, the operation in the `calculateAverage()` method (called at (2)) is dependent on the success of the `sumNumbers()` method (called at (1)). The `if` statement at (2) checks the value of the `sum` variable before calling the `calculateAverage()` method:

Click here to view code image

```
int sum = 0;
try {
  sum = sumNumbers();                  // (1)
  // other actions
} finally {
  if (sum > 0) calculateAverage();      // (2)
}
```

This code guarantees that if the `try` block is entered, the `sumNumbers()` method will be executed first, and later the `calculateAverage()` method will be executed in the `finally` clause, regardless of how execution proceeds in the `try` block. We can, if desired, include any `catch` clauses to handle any exceptions.

If the `finally` clause neither throws an exception nor executes a control transfer statement like a `return` or a labeled `break`, the execution of the `try` block or any `catch` clause determines how execution proceeds after the `finally` clause (Figure 7.4, p. 376).

- If no exception is thrown during execution of the `try` block or the exception has been handled in a `catch` clause, normal execution continues after the `finally` clause.
- If there is any uncaught exception (either because no matching `catch` clause was found or because the `catch` clause threw an exception), the method completes abruptly and the exception is propagated after the execution of the `finally` clause.

The output of Example 7.4 shows that the `finally` clause at (4) is executed, regardless of whether an exception is thrown in the `try` block at (2). If an `Arithmetic-Exception` is thrown, it is caught and handled by the `catch` clause at (3). After the execution of the `finally` clause at (4), normal execution continues at (5).

Example 7.4 *The* `try-catch-finally` *Construct*

Click here to view code image

```
public class Average4 {

  public static void main(String[] args) {
    printAverage(100, 20);                                         // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                          // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                             // (3)
      ae.printStackTrace();
      System.out.println("Exception handled in printAverage().");
    } finally {                                                    // (4)
```

```
        System.out.println("Finally done.");
      }
      System.out.println("Exit printAverage().");                    // (5)
    }

    public static int computeAverage(int sum, int count) {
      System.out.println("Computing average.");
      return sum/count;
    }
  }
```

Output from the program, with the call `printAverage(100, 20)` at (1):

Click here to view code image

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Exit printAverage().
Exit main().
```

Output from the program, with the call `printAverage(100, 0)` at (1):

Click here to view code image

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average4.computeAverage(Average4.java:24)
        at Average4.printAverage(Average4.java:10)
        at Average4.main(Average4.java:4)
Exception handled in printAverage().
Finally done.
Exit printAverage().
Exit main().
```

On exiting from the `finally` clause, if there is any uncaught exception, the method completes abruptly and the exception is propagated as explained earlier. This is illustrated in Example 7.5. The method `printAverage()` is aborted after the `finally` clause at (3) has been executed, as the `ArithmeticException` thrown at (4) is not caught by any method. In this case, the exception is handled by the default exception handler. Notice the difference in the output from Example 7.4 and Example 7.5.

Example 7.5 *The* `try-finally` *Construct*

Click here to view code image

```
public class Average5 {

  public static void main(String[] args) {
    printAverage(100, 0);                                          // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                             // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } finally {                                                       // (3)
      System.out.println("Finally done.");
    }
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;                                                 // (4)
  }
}
```

Output from the program:

Click here to view code image

```
Computing average.
Finally done.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average5.computeAverage(Average5.java:21)
        at Average5.printAverage(Average5.java:10)
        at Average5.main(Average5.java:4)
```

If the `finally` clause executes a control transfer statement, such as a `return` or a labeled `break`, this control transfer statement determines how the execution will proceed—regardless of how the `try` block or any `catch` clause was executed. In particular, a value returned by a `return` statement in the `finally` clause will supercede any value returned by a `return` statement in the `try` block or a `catch` clause.

Example 7.6 shows how the execution of a control transfer statement such as a `return` in the `finally` clause affects the program execution. The first output from the program shows that the average is computed but the value returned is from the `return` statement at (3) in the `finally` clause, not from the `return` statement at (2) in the `try` block. The second output shows that the `ArithmeticException` thrown in the `computeAverage()` method and propagated to the `printAverage()` method is suppressed by the `return` statement in the `finally` clause. Normal execution continues after the `return` statement at (3), with the value `0` being returned from the `printAverage()` method.

If the `finally` clause throws an exception, this exception is propagated with all its ramifications—regardless of how the `try` block or any `catch` clause was executed. In particular, the new exception overrules any previously uncaught exception (p. 415).

Example 7.6 *The* `finally` *Clause and the* `return` *Statement*

Click here to view code image

```
public class Average6 {

  public static void main(String[] args) {
    System.out.println("Value: " + printAverage(100, 20));          // (1)
    System.out.println("Exit main().");
  }

  public static int printAverage(int totalSum, int totalCount) {
    int average = 0;
    try {
      average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
      return average;                                                // (2)
    } finally {
      System.out.println("Finally done.");
      return average*2;                                              // (3)
    }
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;
  }
}
```

Output from the program, with call `printAverage(100, 20)` at (1):

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Value: 10
Exit main().
```

Output from the program, with call `printAverage(100, 0)` at (1):

```
Computing average.
Finally done.
Value: 0
Exit main().
```

7.4 The `throw` Statement

Earlier examples in this chapter have shown how an exception can be thrown implicitly by the JVM during execution. Now we look at how an application can programmatically throw an exception using the `throw` statement. This statement can be used in a method, a constructor, or an initializer block. The general format of the `throw` statement is as follows:

Click here to view code image

```
throw object_reference_expression;
```

The compiler ensures that the type of the *object reference expression* is a `Throwable` or one of its subclasses. This ensures that a `Throwable` will always be propagated. At runtime a `NullPointerException` is thrown by the JVM if the *object reference expression* evaluates to `null`.

A detail message is often passed to the constructor when the exception object is created.

Click here to view code image

```
throw new ArithmeticException("Integer division by 0");
```

Propagation of a programmatically thrown exception is no different from one thrown implicitly by the JVM. When an exception is thrown, normal execution is suspended. The JVM proceeds to find a `catch` clause that can handle the exception.

The search starts in the context of the current `try` block, propagating to any enclosing `try` blocks and through the JVM stack to find a handler for the exception. Any associated `finally` clause of a `try` block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level. If a handler is found, normal execution resumes after the code in its `catch` clause has been executed, barring any rethrowing of an exception.

In Example 7.7, an exception is thrown by the `throw` statement at (4) as the `count` value is `0`. This exception is propagated to the `printAverage()` method, where it is caught and handled by the `catch` clause at (1). Note that the `finally` clause at (2) is executed, followed by the resumption of normal execution, as evident from the output of the print statement at (3).

Example 7.7 *Throwing Exceptions Programmatically*

Click here to view code image

```
public class Average7 {

  public static void main(String[] args) {
    printAverage(100, 0);            // Calling with 0 number of values.
  }
  public static void printAverage(int totalSum, int totalCount) {
    System.out.println("Entering printAverage().");
    try {
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                             // (1)
      ae.printStackTrace();
      System.out.println("Exception handled in printAverage().");
    } finally {                                                    // (2)
      System.out.println("Finally in printAverage().");
    }
    System.out.println("Exit printAverage().");                    // (3)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    if (count == 0)
      throw new ArithmeticException("Integer division by 0");      // (4)
    return sum/count;
  }
}
```

Output from the program:

Click here to view code image

```
Entering printAverage().
Computing average.
java.lang.ArithmeticException: Integer division by 0
      at Average7.computeAverage(Average7.java:26)
      at Average7.printAverage(Average7.java:11)
      at Average7.main(Average7.java:5)
Exception handled in printAverage().
Finally in printAverage().
Exit printAverage().
```

7.5 The `throws` Clause

A `throws` clause can be specified in a method or a constructor header to declare any checked exceptions that can be thrown by a statement in the body of a method or a constructor. It is declared immediately preceding the body of the method or the constructor.

Click here to view code image

`... throws` *ExceptionType*$_1$`,` *ExceptionType*$_2$`,...,` *ExceptionType*$_n$ `{ /* Body */ }`

Each *ExceptionType*$_i$ is an *exception type* (i.e., a `Throwable` or one of its subclasses), although usually only checked exceptions are specified. The compiler enforces that if a checked exception can be thrown from the body of the method or the constructor, then either the type of this exception or a supertype of its exception type is specified in the `throws` clause of the method or the constructor. The `throws` clause can specify unchecked exceptions, but this is seldom done and the compiler does not enforce any restrictions on their usage.

The `throws` clause is part of the contract that a method or a constructor offers to its clients. The `throws` clause can specify any number of exception types in any order, even those that are not thrown by the method or the constructor. The compiler simply ensures that any checked exception that can actually be thrown in the method or constructor body is covered by the `throws` clause. Of course, any caller of the method or constructor cannot ignore the checked exceptions specified in the `throws` clause.

In a method or a constructor, a checked exception can be thrown directly by a `throw` statement, or indirectly by calling other methods or constructors that can throw a checked exception. If a checked exception is thrown, the code must obey the following rule (known by various names: *catch-or-declare rule, handle-or-declare rule, catch-or-specify requirement*):

- Either use a `try` block and catch the checked exception in a `catch` block and deal with it
- Or explicitly allow propagation of the checked exception to its caller by declaring it in the `throws` clause

Note that catching and dealing with a checked exception does not necessarily imply resumption of normal execution. A `catch` clause can catch the checked exception and choose to throw some other exception or even the same exception that is either unchecked or declared in the `throws` clause (p. 401). This rule ensures that a checked exception will be dealt with, regardless of the path of execution. This aids development of robust programs, as allowance can be made for many contingencies.

In Example 7.8, a new checked exception is defined, where the checked exception class `IntegerDivisionByZero` extends the `Exception` class. The method call at (2) in the `try` block at (1) results in the `printAverage()` method at (6) to be executed. The method call at (7) results in the `computeAverage()` method at (8) to be executed.

In the `if` statement at (9), the method `computeAverage()` throws the checked exception `IntegerDivisionByZero`. Neither the `computeAverage()` method nor the `printAverage()` method catches the exception, but instead throws it to the caller, as declared in the `throws` clauses in their method headers at (6) and (8). The exception propagates to the `main()` method. Since the `printAverage()` method was called from the context of the `try` block at (1) in the `main()` method, the exception is successfully caught by its `catch` clause at (3). The exception is handled and the `finally` clause at (4) is executed, with normal execution resuming from (5). If the method `main()` did not catch the exception, it would have to declare this exception in a `throws` clause. In that case, the exception would end up being handled by the default exception handler.

Example 7.8 *The* `throws` *Clause*

Click here to view code image

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

Click here to view code image

```
// File: Average8.java
public class Average8 {
  public static void main(String[] args) {
    try {                                                          // (1)
      printAverage(100, 0);                                        // (2)
    } catch (IntegerDivisionByZero idbz) {                         // (3)
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                    // (4)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                            // (5)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {                               // (6)
    int average = computeAverage(totalSum, totalCount);            // (7)
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {                               // (8)
    System.out.println("Computing average.");
    if (count == 0)                                                // (9)
      throw new IntegerDivisionByZero();
    return sum/count;                                              // (10)
  }
}
```

Output from the program:

Click here to view code image

```
Computing average.
IntegerDivisionByZero: Integer Division By Zero
        at Average8.computeAverage(Average8.java:27)
        at Average8.printAverage(Average8.java:17)
        at Average8.main(Average8.java:5)
Exception handled in main().
Finally done in main().
Exit main().
```

As mentioned earlier, the exception type specified in the `throws` clause can be a superclass of the actual exceptions thrown—that is, the exceptions thrown must be assignable to the type of the exceptions specified in the `throws` clause. If a method or a constructor can throw a checked exception, then the `throws` clause must declare its exception type or a supertype of its exception type; otherwise, a compile-time error will occur. In the `printAverage()` method, the superclass `Exception` of the subclass `IntegerDivisionByZero` could be specified in the `throws` clause of the method. This would also entail that the `main()` method either catch an `Exception` or declare it in a `throws` clause.

Click here to view code image

```
public static void main(String[] args) throws Exception {
  /* ... */
}
public static void printAverage(int totalSum, int totalCount) throws Exception
  /* ... */
}
```

It is generally considered bad programming style to specify exception superclasses in the `throws` clause when the actual exceptions thrown are instances of their subclasses. It is also recommended to use the `@throws` tag in a Javadoc comment to document the checked exceptions that a method or a constructor can throw, together with any unchecked exceptions that might also be relevant to catch.

Overriding the `throws` Clause

A subclass can *override* a method defined in its superclass by providing a new implementation (§5.1, p. 196). What happens when a superclass method with a list of exceptions in its `throws` clause is overridden in a subclass? The method declaration in the subclass need not specify a `throws` clause if it does not throw any checked exceptions, and if it does, it can specify only *checked* exception classes that are already in the `throws` clause of the superclass method, or that are subclasses of the checked exceptions in the `throws` clause of the superclass method. As a consequence, an overriding method can have more number of exceptions, but it *cannot* allow *broader* checked exceptions in its `throws` clause than the superclass method does. Allowing broader checked exceptions in the overriding method would create problems for clients who already deal with the exceptions specified in the superclass method. Such clients would be ill prepared if an object of the subclass threw a checked exception they were not prepared for. However, there are no restrictions on specifying *unchecked* exceptions in the `throws` clause of the overriding method. The preceding discussion also applies to overriding methods from an interface that a class implements.

In the code below, the method `compute()` at (1) in superclass `A` is overridden correctly at (2) in subclass `B`. The `throws` clause of the method at (2) in subclass `B` specifies only one checked exception (`ThirdException`) from the `throws` clause at (1) and adds the more specific subclass exception (`SubFirstException`) of the superclass exception (`FirstException`) that is specified in the `throws` clause at (1). An unchecked exception (`NumberFormatException`) is also specified in the throws clause at (2). The checked exceptions in the `throws` clause at (2) are covered by the checked exceptions specified in the `throws` clause at (1). Unchecked exceptions are inconsequential in this regard. The subclass `C` does not override the `compute()` method from class `A` correctly, as the `throws` clause at (3) specifies an exception (`FourthException`) that the overridden method at (1) in class `A` cannot handle.

Click here to view code image

```
// New exception classes:
class FirstException    extends Exception { }
class SecondException   extends Exception { }
class ThirdException    extends Exception { }
class FourthException   extends Exception { }
class SubFirstException extends FirstException { }

// Superclass
class A {
  protected void compute()
      throws FirstException, SecondException, ThirdException { /* ... */ }  //
}

// Subclass
class B extends A {
  @Override
  protected void compute()
      throws ThirdException, SubFirstException, NumberFormatException {      //
    /* ... */
  }
}
//Subclass
class C extends A {
    @Override
    protected void compute()                                   // Compile-time error at
        throws FirstException, ThirdException, FourthException { /* ... */ }  //
  }
```

Usage of checked and unchecked exceptions in different contexts is compared in Table 7.1.

Table 7.1 *Comparing Checked and Unchecked Exceptions*

Context	Checked exceptions	Unchecked exceptions
`throws` clause	Can include any checked exception, but when overridden cannot specify new checked exceptions.	Can include any unchecked exception, whether it is overridden or not.
`try` block	Can throw any checked exception.	Can throw any unchecked exception.
`catch` clause	Can only catch a checked exception that is thrown by the `try` block. A catch clause can always be used to catch an `Exception`.	Can catch any unchecked exception, whether it is thrown by the `try` block or not.
`throw` statement	Can throw any checked exception.	Can throw any unchecked exception.
Catch-or-declare policy	Applies only to checked exceptions.	Does not apply to unchecked exceptions.

Review Questions

7.1 Which digits, and in which order, will be printed when the following program is run?

Click here to view code image

```
public class DemoClass {
  public static void main(String[] args) {
    int k=0;
    try {
      int i = 5/k;
    } catch (ArithmeticException e) {
      System.out.println("1");
    } catch (RuntimeException e) {
      System.out.println("2");
      return;
    } catch (Exception e) {
      System.out.println("3");
    } finally {
      System.out.println("4");
    }
    System.out.println("5");
  }
}
```

Select the one correct answer.

a. The program will only print `5`.

b. The program will only print `1` and `4`, in that order.

c. The program will only print `1`, `2`, and `4`, in that order.

d. The program will only print `1`, `4`, and `5`, in that order.

e. The program will only print `1`, `2`, `4`, and `5`, in that order.

f. The program will only print `3` and `5`, in that order.

7.2 Which of the following statements are true about the following program?

Click here to view code image

```
public class Exceptions {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    } finally {
      System.out.println("The end");
    }
  }
}
```

Select the two correct answers.

a. If run with no program arguments, the program will produce no output.

b. If run with no program arguments, the program will print `The end`.

c. The program will throw an `ArrayIndexOutOfBoundsException`.

d. If run with one program argument, the program will simply print the specified argument.

e. If run with one program argument, the program will print the specified argument followed by `The end`.

7.3 Which of the following statements are true? Select the two correct answers.

a. If an exception is not caught in a method, the method will terminate and normal execution will resume.

b. An overriding method must declare that it throws the same exception classes as the method it overrides.

c. The `main()` method of a program can declare that it throws checked exceptions.

d. A method declaring that it throws an exception of a certain class may throw instances of any subclass of that exception class.

e. The `finally` clause is executed if, and only if, an exception is thrown in the corresponding `try` block.

7.4 Which statement is true about the following code?

Click here to view code image

```
class A extends Throwable {}

class B extends A {}

public class RQ6A20 {
  public static void main(String[] args) throws A {
    try {
      action();
    } finally {
      System.out.println("Done.");
    } catch (A e) {

      throw e;
    }
  }

  public static void action() throws B {
    throw new B();
  }
}
```

Select the one correct answer.

- **a.** The `main()` method must declare that it throws `B`.
- **b.** The `finally` clause must follow the `catch` clause in the `main()` method.
- **c.** The `catch` clause in the `main()` method must declare that it catches `B` rather than `A`.
- **d.** A single `try` block cannot be followed by both `catch` and `finally` clauses.
- **e.** The declaration of class `A` is not valid.

7.5 Which statement is true about the following code?

Click here to view code image

```
public class Calculator {
  public static int average(int... values) {
    int result = 0;
    for (int i = 0; i < values.length; i++) {
      result += values[i];
    }
    return result/values.length;
  }
  public static void main(String[] args) {
    int value = 1;
    try {
      value = average();
    } catch (ArithmeticException e) {
      System.out.print("error");
    }
    System.out.print(value);
  }
}
```

Select the one correct answer.

- **a.** The program will print `error0`.
- **b.** The program will print `error1`.
- **c.** The program will print `error`.
- **d.** The program will print `0`.

e. The program will print `1`.

f. The program will terminate by throwing an exception.

g. The program will fail to compile.

7.6 What will be the output of the following program?

Click here to view code image

```
public class PlayerException extends Exception {
  public PlayerException(String message) { super(message); }
}
```

Click here to view code image

```
public class Player {
  public static String reaction (String action)  throws PlayerException
    if (action == null || action.isEmpty()) {
      throw new PlayerException("Invalid action");
    }
    return ">" + action;
  }
  public static void main(String[] args) {
    String message = null;
    try {
      message = reaction(message);
    } catch (PlayerException e) {
      message = e.getMessage();
    }
    System.out.print(message);
  }
}
```

Select the one correct answer.

a. `message`

b. `>message`

c. `>null`

d. `>`

e. `Invalid action`

f. `>Invalid action`

7.7 What will be the output of the following program?

Click here to view code image

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws Exception {

    if (name == null) throw new FileNotFoundException("invalid file name
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

a. `IO error: invalid file name finally`
b. `IO error: invalid file name the end`
c. `IO error: invalid file name finally the end`
d. `Other error: invalid file name finally`
e. `Other error: invalid file name the end`
f. `Other error: invalid file name finally the end`
g. `finally the end`
h. `the end`
i. Nothing will be printed.
j. An uncaught exception stack trace will be printed.

7.8 What will be the output of the following program?

Click here to view code image

```
import java.io.*;
public class FileReader {
  public static void readFile(String name)  throws Exception {

    if (name != null) throw new FileNotFoundException("invalid file name
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage()+" ");
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage()+" ");
    } finally {
      System.out.print("finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

a. `IO error: invalid file name finally`
b. `IO error: invalid file name the end`
c. `IO error: invalid file name finally the end`
d. `Other error: invalid file name finally`
e. `Other error: invalid file name the end`
f. `Other error: invalid file name finally the end`
g. `finally the end`
h. `the end`
i. Nothing will be printed.
j. An uncaught exception stack trace will be printed.

7.9 What will be the output of the following program?

Click here to view code image

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws IOException {

    if (name == null) {
      throw new NullPointerException("invalid file name");
    } else {
      throw new IOException("file read not implemented");
    }
  }

  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
      return;
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

a. `IO error: file read not implemented finally`

b. `IO error: file read not implemented the end`

c. `IO error: file read not implemented finally the end`

d. `Other error: invalid file name finally`

e. `Other error: invalid file name the end`

f. `Other error: invalid file name finally the end`

g. `finally the end`

h. `the end`

i. Nothing will be printed.

j. An uncaught exception stack trace will be printed.

7.6 The Multi-`catch` Clause

Example 7.9 uses a `try` block that has multiple uni-`catch` clauses. This example is based on Example 7.8. The sum of the values and the number of values needed to calculate the average are now read as program arguments from the command line at (2) and (3), respectively. The example shows a `try` statement at (1) that uses three uni-`catch` clauses: at (5), (6), and (7). In a *uni*-`catch` clause, a *single* exception type is specified for the `catch` parameter.

In Example 7.9, the method `printAverage()` is only called at (4) if there are at least two consecutive integers specified on the command line. An unchecked `ArrayIndexOutOfBoundsException` is thrown if there are not enough program arguments, and an unchecked `NumberFormatException` is thrown if an argument cannot be converted to an `int` value. The astute reader will notice that the code for handling these two exceptions is the same in the body of the respective `catch` clauses. In order to avoid such code duplication, one might be tempted to replace the two `catch` clauses with a single `catch` clause that catches a more *general* exception, for example:

Click here to view code image

```
catch (RuntimeException rte) { // NOT RECOMMENDED!
  System.out.println(rte);
  System.out.println("Usage: java Average9 <sum of values> <no. of values>");
}
```

This is certainly not recommended, as specific exceptions are to be preferred over general exceptions, not the least because a more general exception type might unintentionally catch more exceptions than intended.

Example 7.9 *Using Multiple* `catch` *Clauses*

Click here to view code image

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

Click here to view code image

```
// File: Average9.java
public class Average9 {
  public static void main(String[] args) {
    try {                                                     // (1)
      int sum         = Integer.parseInt(args[0]);            // (2)
      int numOfValues = Integer.parseInt(args[1]);            // (3)
      printAverage(sum, numOfValues);                         // (4)
    } catch (ArrayIndexOutOfBoundsException aioob) {          // (5) uni-catch
      System.out.println(aioob);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>
    } catch (NumberFormatException nfe) {                     // (6) uni-catch
      System.out.println(nfe);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>
    } catch (IntegerDivisionByZero idbz) {                    // (7) uni-catch
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                               // (8)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                       // (9)
  }
  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    int average = computeAverage(totalSum, totalCount);
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    System.out.println("Computing average.");
    if (count == 0)
      throw new IntegerDivisionByZero();
    return sum/count;
  }
}
```

Running the program:

Click here to view code image

```
>java Average9 100 twenty
java.lang.NumberFormatException: For input string: "twenty"
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

Running the program:

Click here to view code image

```
>java Average9 100
java.lang.ArrayIndexOutOfBoundsException: 1
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

The *multi*-`catch` clause provides the solution, allowing specific exceptions to be declared and avoiding duplicating the same code for the body of the `catch` clauses. The syntax of the multi-`catch` clause is as follows:

Click here to view code image

```
catch (exception_type1|exception_type2|...|exception_typek parameter) {
  statements
}
```

The multi-`catch` clause still has a single *parameter*, but now a list of exception types, delimited by the vertical bar (|), can be specified as the types for this parameter. This list defines a *union of alternatives* that are the exception types which the multi-`catch` clause can handle. The *statements* in the body of the multi-`catch` clause will be executed when an object of any of the specified exception types is caught by the multi-`catch` clause.

The multiple `catch` clauses at (5) and (6) in Example 7.9 have been replaced with a multi-`catch` clause at (5) in Example 7.10:

Click here to view code image

```
catch (ArrayIndexOutOfBoundsException |                   // (5) multi-catch
       NumberFormatException ep) {
  System.out.println(ep);
  System.out.println("Usage: java Average10 <sum of values> <no. of values>");
}
```

The multi-`catch` clause in Example 7.10 is semantically equivalent to the two uni-`catch` clauses in Example 7.9, and we can expect the same program behavior in both examples.

Example 7.10 *Using the Multi-*`catch` *Clause*

Click here to view code image

```
// File: Average10.java
public class Average10 {
  public static void main(String[] args) {
    try {                                                        // (1)
      int sum = Integer.parseInt(args[0]);                       // (2)
      int numOfValues = Integer.parseInt(args[1]);               // (3)
      printAverage(sum, numOfValues);                            // (4)
    } catch (ArrayIndexOutOfBoundsException |                    // (5) multi-ca
             NumberFormatException ep) {
      System.out.println(ep);
      System.out.println("Usage: java Average10 <sum of values> <no. of values
    } catch (IntegerDivisionByZero idbz) {                       // (6) uni-catc
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                  // (7)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                          // (8)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }
}
```

A few remarks are in order regarding the alternatives of a multi-`catch` clause. There should be no subtype–supertype relationship between any of the specified exception types in the alternatives of a multi-`catch` clause. The following multi-`catch` clause will not compile, as `ArrayIndexOutOfBoundsException` is a subtype of `IndexOutOfBoundsException`:

Click here to view code image

```
catch (IndexOutOfBoundsException |                   // Compile-time error!
         ArrayIndexOutOfBoundsException e) {
    // ...
  }
```

The parameter of a multi-`catch` clause is also considered to be *implicitly* `final`, and therefore cannot be assigned to in the body of the multi-`catch` clause. In a uni-`catch` clause, the parameter is considered to be *effectively* `final` if it does not occur on the left-hand side of an assignment in the body of the uni-`catch` clause.

Click here to view code image

```
  try {
    // Assume appropriate code to throw the right exceptions.
  } catch (NumberFormatException |
           IndexOutOfBoundsException e) {    // Parameter is final.
    e = new ArrayIndexOutOfBoundsException();// Compile-time error!
                                             // Cannot assign to final parameter
  } catch (IntegerDivisionByZero idbz) {     // Parameter is effectively final.
    idbz.printStackTrace();
  } catch (IOException ioe) {                // Parameter is not effectively fin
    ioe = new FileNotFoundException("No file.");
  }
```

Disallowing any subtype–supertype relationship between alternatives and the parameter being `final` in a multi-`catch` clause or effectively `final` in a uni-`catch` clause allows the compiler to perform precise exception handling analysis.

The compiler also generates effective bytecode for a *single* exception handler corresponding to all the alternatives in a multi-`catch` clause, in contrast to generating bytecode for *multiple* exception handlers for uni-`catch` clauses that correspond to the multi-`catch` clause.

Rethrowing Exceptions

Rethrowing an exception refers to throwing an exception in the body of a `catch` clause. The `catch` clause catches an exception, but then throws this exception or another exception in its body. This allows an exception to be partially handled when it is caught the first time, and then again when the rethrown exception is caught later. Typically, the first exception handler is a common handler for the situation and the later exception handler is a more specific one.

Exception parameters that are explicitly, implicitly, or effectively `final` in `catch` clauses allow the compiler to perform improved analysis of exception handling in the code, especially when it comes to rethrowing exceptions.

For the examples in this section, it is important to keep in mind that the exception type `IOException` is the supertype of both `EOFException` and `FileNotFoundException`.

Example 7.11 illustrates how the compiler is able to identify unreachable code by precise analysis of rethrown exceptions that are either `final` or effectively `final`. The body of the `try` statement at (1) can only throw a `FileNotFoundException` that is caught by the `catch` clause at (3). This exception is effectively `final` in the `catch` clause at (3), as no assignment is made to it in the body of the `catch` clause. This exception is rethrown in the nested `try` statement at (4), but the `catch` clause at (6) of this `try` statement can only catch an `EOFException`. Since parameter `ex` is effectively `final`, it can only denote a `FileNotFoundException`, never an `EOFException`. The `catch` clause at (6) is unreachable, and the compiler flags an error.

Example 7.11 *Precise Rethrowing of Exceptions*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionAnalysis {
  public static void main(String[] args) throws IOException {
    try {                                // (1)
      throw new FileNotFoundException(); // (2)
    } catch (IOException ex) {           // (3)
      try {                        // (4) Nested try statement
        throw ex;                  // (5) Can only rethrow FileNotFoundExcepti
      } catch (EOFException se) {  // (6) Compile-time error: clause unreachab
        System.out.println("I am unreachable.");
      }

    }
  }
}
```

Example 7.12 illustrates how `final` or effectively `final` `catch` parameters allow more precise exceptions to be specified in the `throws` clause of a method. The thing to note is that the parameter `e` in the `catch` clause at (5) is not effectively `final`, as an assignment is made to the parameter at (6). All bets are off when the parameter is not `final` and the exception is rethrown. The `throws` clause must specify the same type as the type of the `catch` parameter, as shown at (3a). This has consequences for the caller method `main()`. Its `try` statement at (1) must include the `catch` clause at (2) to catch an `IOException` as well, or the compiler will flag an error about an uncaught checked exception. The contract of the `checkIt()` method allows for all exceptions that are either

`IOException` or its subtypes.

If the assignment statement at (6) is commented out, the `catch` parameter `e` is effectively `final` in the `catch` body. The compiler can deduce that the rethrown exception can only be of type `FileNotFoundException` or `EOFException`. The `throws` clause of the `checkIt()` method can be made more specific, as at (3b). Note that the type of the `catch` parameter is the supertype `IOException` of the subtypes specified in the `throws` clause at (3b). Commenting out the assignment statement at (6) and uncommenting the more precise `throws` clause at (3b) has consequences for the caller method `main()` as well. The `catch` clause at (2) becomes unreachable, and the compiler issues a *warning*. Note also that the type of the exception parameter `e` at (5) is `IOException`, which is the supertype of the exception types specified in the `throws` clause. However, static analysis by the compiler is able to confirm that the exception parameter `e` can only denote objects of either `FileNotFoundException` or `EOFException`, but not of supertype `IOException`.

Example 7.12 *Precise* `throws` *Clause*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class MorePreciseRethrow {
  public static void main(String[] args) {                         // (1)
    try {
      checkIt(1);
    } catch (FileNotFoundException fnfe) {
      System.out.println("Check that the file exits.");
    } catch (EOFException eofe) {
      System.out.println("Check the contents of the file.");
    } catch (IOException ioe) { // (2) mandatory with (3a), but compiler warni
                                //     that clause is unreachable with (3b).
      System.out.println("This should never occur.");
    }
  }

  public static void checkIt(int value) throws IOException {       // (3a)
//public static void checkIt(int value)                            // (3b)
//    throws FileNotFoundException, EOFException {
    try {                                                          // (4)

      switch (value) {
        case 1:
          throw new FileNotFoundException("File not found");
        case 2:
          throw new EOFException("End of file");
        default:
          System.out.println("OK");
      }
    } catch (IOException e) {                                       // (5)
      System.out.println(e.getMessage());
      e = new EOFException("End of file");          // (6) not effectively fina
                                                    //     requires (3a).
                                                    //     When commented out,
                                                    //     can use (3b).
      throw e;
    }
  }
}
```

Program output with (3a) and (6) uncommented, and (3b) commented out:

Click here to view code image

```
File not found
Check the contents of the file.
```

Program output with (3a) and (6) commented out, and (3b) is uncommented:

Click here to view code image

```
File not found
Check that the file exits.
```

In summary, a `throw` statement in the body of a `catch` clause can throw a `final` or an effectively `final` exception parameter that has exception type `E` if *all* of the following conditions are satisfied:

- Exception type `E` can be thrown in the body of the `try` statement with which the `catch` clause is associated.
- Exception type `E` is assignment compatible with any of the exception types declared for the parameter in the `catch` clause.
- Exception type `E` is not assignment compatible with any of the exception types declared for the parameters in any preceding `catch` clause in the same `try` statement.

In Example 7.13, the `throw` statements at (1), (2), and (3) all try to rethrow an exception that is effectively `final` in the body of the `catch` clause.

- The `throw` statement at (1) in the `main()` method satisfies all the conditions. The `try` block throws an exception of the right type (`EOFException`). The exception thrown (`EOFException`) is assignment compatible with the type declared for the parameter in the `catch` clause (`IOException`). There is no preceding `catch` clause that handles the exception (`EOFException`).

- The `throw` statement at (2) in the `rethrowA()` method cannot throw the exception, as the first condition is not satisfied: The `try` block does not throw an exception of the right type (`EOFException`). The compiler flags an *error* for the `catch` clause that is unreachable.
- The `throw` statement at (3) in the `rethrowB()` method cannot throw the exception, as the third condition is not satisfied: A preceding `catch` clause can handle the exception (`EOFException`). The compiler flags a *warning* for the `catch` clause which is unreachable.

Example 7.13 *Conditions for Rethrowing Final Exceptions*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class RethrowMe {
  public static void main(String[] args) throws EOFException {
    try {
      switch (1) {
        case 1: throw new FileNotFoundException("File not found");
        case 2: throw new EOFException("End of file");
        default: System.out.println("OK");
      }
    } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe);
    } catch (IOException ioe) {
      throw ioe;                                                    // (1)
    }
  }

  public static void rethrowA() throws EOFException {
    try {
      // Empty try block.
    } catch (EOFException eofe) { // Compile-time error: exception not thrown
                                  //                     in try block.
      throw eofe;                                                   // (2)
    }
  }

  public static void rethrowB() throws EOFException {
    try {
      throw new EOFException("End of file");
    } catch (EOFException eofe) {
      System.out.println(eofe);
    } catch (IOException ioe) {   // Compile-time warning: unreachable clause
      throw ioe;                                                    // (3)
    }
  }
}
```

Chaining Exceptions

It is common that the handling of an exception leads to the throwing of another exception. In fact, the first exception is the cause of the second exception being thrown. Knowing the *cause* of an exception can be useful, for example, when debugging the application. The Java API provides a mechanism for *chaining* exceptions for this purpose.

The class `Throwable` provides the following constructors and methods to handle chained exceptions:

> Click here to view code image
>
> ```
> Throwable(Throwable cause)
> Throwable(String msg, Throwable cause)
> ```
>
> Sets the throwable to have the specified cause, and also specifies a detail message if the second constructor is used. Most exception types provide analogous constructors.
>
> Click here to view code image
>
> ```
> Throwable initCause(Throwable cause)
> ```
>
> Sets the cause of this throwable. Typically called on exception types that do not provide a constructor to set the cause.
>
> ```
> Throwable getCause()
> ```
>
> Returns the cause of this throwable, if any; otherwise, it returns `null`.

Example 7.14 illustrates how a chain of cause-and-effect exceptions, referred to as the *backtrace,* associated with an exception can be created and manipulated. In the method `chainIt()`, declared at (2), an exception is successively caught and associated as a *cause* with a new exception before the new exception is thrown, resulting in a chain of exceptions. This association is made at (3) and (4). The `catch` clause at (1) catches the exception thrown by the method `checkIt()`. The causes in the chain are successively retrieved by calling the `getCause()` method. Program output shows the resulting backtrace: which exception was the cause of which exception, and the order shown being reverse to the order in which they were thrown, the first one in the chain being thrown last.

Example 7.14 *Chaining Exceptions*

Click here to view code image

```
import java.io.EOFException;
```

```
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionsInChain {
  public static void main(String[] args) {
    try {
      chainIt();
    } catch (Exception e) {                                  // (1)
      System.out.println("Exception chain: " + e);
      Throwable t = e.getCause();
      while (t != null) {
        System.out.println("Cause: " + t);
        t = t.getCause();
      }
    }
  }

  public static void chainIt() throws Exception {            // (2)
    try {
      throw new FileNotFoundException("File not found");
    } catch (FileNotFoundException e) {
      try {
        IOException ioe = new IOException("File error");
        ioe.initCause(e);                                    // (3)
        throw ioe;
      } catch (IOException ioe) {
        Exception ee = new Exception("I/O error", ioe);      // (4)
        throw ee;
      }
    }
  }
}
```

Output from the program:

Click here to view code image

```
Exception chain: java.lang.Exception: I/O error
Cause: java.io.IOException: File error
Cause: java.io.FileNotFoundException: File not found
```

A convenient way to print the backtrace associated with an exception is to invoke the `printStackTrace()` method on the exception. The `catch` clause at (1) in Example 7.14 can be replaced with the `catch` clause at (1') below that calls the `printStackTrace()` method on the exception.

Click here to view code image

```
...
try {
  chainIt();
} catch (Exception e) {                              // (1')
  e.printStackTrace();                               // Print backtrace.
}
...
```

The refactoring of Example 7.14 will result in the following analogous printout of the backtrace, showing as before the exceptions and their causes in the reverse order to the order in which they were thrown:

Click here to view code image

```
java.lang.Exception: I/O error
        at ExceptionsInChain.chainIt(ExceptionsInChain.java:23)
        at ExceptionsInChain.main(ExceptionsInChain.java:8)
Caused by: java.io.IOException: File error
        at ExceptionsInChain.chainIt(ExceptionsInChain.java:19)
        ... 1 more
Caused by: java.io.FileNotFoundException: File not found
        at ExceptionsInChain.chainIt(ExceptionsInChain.java:16)
        ... 1 more
```

7.7 The `try`-with-resources Statement

Normally, objects in Java are automatically garbage collected at the discretion of the JVM when they are no longer in use. However, *resources* are objects that need to be explicitly closed when they are no longer needed. Files, streams, and database connections are all examples that fall into this category of objects. Such resources also rely on underlying system resources for their use. By closing such resources, any underlying system resources are also freed, and can therefore be recycled. Resource leakage (i.e., failure to close resources properly) can lead to performance degradation as resources get depleted.

Best practices recommend the following idiom for using a resource:

- Open the resource—that is, allocate or assign the resource.
- Use the resource—that is, call the necessary operations on the resource.
- Close the resource—that is, free the resource.

Typically, all three steps above can throw exceptions, and to ensure that a resource is always closed after use, the recommended practice is to employ a combination of `try-catch-finally` blocks. The first two steps are usually nested in a `try` block, with any associated `catch` clauses, and the third step is executed in a `finally` clause to ensure that the resource, if it was opened, is always closed regardless of the path of execution through the `try-catch` blocks.

Example 7.15 shows a naive approach to resource management, that can result in resource leakage. It uses a `BufferedReader` associated with a `FileReader` to read a line from a text file (§20.3, p. 1251). The example follows the idiom for resource usage. As we can see, an exception can be thrown from each of the steps. If any of the first two steps throw an exception, the call to the `close()` method at (4) will never be executed to close the resource and thereby any underlying resources, as execution of the method will be terminated and the exception propagated.

Example 7.15 *Naive Resource Management*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class NaiveResourceUse {
  public static void main(String[] args)
      throws FileNotFoundException, EOFException, IOException {

    // Open the resource:
    var fis = new FileReader(args[0]);              // (1) FileNotFoundExceptio
    var br = new BufferedReader(fis);

    // Use the resource:
    String textLine = br.readLine();                // (2) IOException

    if (textLine != null) {
      System.out.println(textLine);
    } else {
      throw new EOFException("Empty file.");        // (3) EOFException
    }

    // Close the resource:
    System.out.println("Closing the resource.");
    br.close();                                     // (4) IOException
  }
}
```

Running the program:

Click here to view code image

```
>java NaiveResourceUse EmptyFile.txt
Exception in thread "main" java.io.EOFException: Empty file.
      at NaiveResourceUse.main(NaiveResourceUse.java:20)
```

Running the program:

Click here to view code image

```
>java NaiveResourceUse Slogan.txt
Code Compile Compute
Closing the resource.
```

Example 7.16 improves on Example 7.15 by explicitly using `try-catch-finally` blocks to manage the resources. No matter how the `try-catch` blocks execute, the `finally` block at (4) will always be executed. If the resource was opened, the `close()` method will be called at (5). The `close()` method is only called on the `BufferedReader` object, and not on the `FileReader` object. The reason is that the `close()` method of the `BufferedReader` object implicitly calls the `close()` method of the `FileReader` object associated with it. This is typical of how the `close()` method of such resources works. Also, calling the `close()` method on a resource that has already been closed normally has no effect; the `close()` method is said to be *idempotent*.

Example 7.16 *Explicit Resource Management*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TryWithoutARM {
  public static void main(String[] args) {
    BufferedReader br = null;
    try {
      // Open the resource:
      var fis = new FileReader(args[0]);             // (1) FileNotFoundExcept
      br = new BufferedReader(fis);

      // Use the resource:
      String textLine = br.readLine();               // (2) IOException

      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");       // (3) EOFException
      }
    } catch (FileNotFoundException | EOFException e) {
      e.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    } finally {                                      // (4)
      if (br != null) {
        try {
          System.out.println("Closing the resource.");
          br.close();                                // (5) IOException
        } catch(IOException ioe) {
          ioe.printStackTrace();
        }
      }
    }
  }
}
```

Running the program:

Click here to view code image

```
>java TryWithoutARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithoutARM.main(TryWithoutARM.java:20)
Closing the resource.
```

Running the program:

Click here to view code image

```
>java TryWithoutARM Slogan.txt
Code Compile Compute
Closing the resource.
```

A lot of boilerplate code is required in explicit resource management using `try-catch-finally` blocks, and the code can get tedious and complex, especially if there are several resources that are open and they all need to be closed explicitly. The `try`-with-resources statement takes the drudgery out of associating `try` blocks with corresponding `finally` blocks to ensure proper resource management. Any resource declared in the header of the `try`-with-resources statement will be automatically closed, regardless of how execution proceeds in the `try` block. The compiler expands the `try`-with-resources statement (and any associated `catch` or `finally` clauses) into basic `try-catch-finally` blocks to guarantee this behavior. It implicitly generates a `finally` block that calls the `close()` method of each resource declared in the header of the `try`-with-resources statement. There is no need to call the `close()` method of the resource, let alone provide any explicit `finally` clause for this purpose.

The `close()` method provided by the resources declared in a `try`-with-resources statement is the implementation of the sole `abstract` method declared in the `java.lang.AutoCloseable` interface. In other words, these resources must implement the `AutoCloseable` interface (p. 412).

The syntax of the `try`-with-resources statement augments the `try` statement with a `try` *header* that comprises a list of *resource declaration statements* separated by a semicolon (`;`).

Click here to view code image

```
try ( resource_declaration_statement1; ... ; resource_declaration_statementm ) {
  statements
} catch (exception_type1 |...| exception_typek parameter1) { // multi-catch
  statements
}
```

```
...
  catch (exception_type_n parameter_n) {                  // uni-catch
    statements
} finally {                                              // finally
    statements
}
```

The syntax shown above is for the *extended* `try`-with-resources statement, which has explicit `catch` and/or `finally` clauses associated with it. The *basic* `try`-with-resources statement has no explicit `catch` or `finally` clauses associated with it.

We will use Example 7.17 to illustrate the salient features of the `try`-with-resources statement. The extended `try`-with-resources statement at (1) declares and initializes two resources in its `try` header:

Click here to view code image

```
try (var fis = new FileReader(args[0]);        // (1) FileNotFoundException
     var br = new BufferedReader(fis)) {
  // ...
}
```

Note that no calls should be made to the `close()` method of a resource variable that is declared and initialized in the `try` header. Any `catch` clause or `finally` clauses associated with an extended `try`-with-resources statement will be executed *after* all the declared resources have been closed. When running Example 7.17 with an empty file, the `EOFException` is caught and handled *after* the two resources denoted by the references `br` and `fis` have been closed.

The scope of a resource variable is the `try` block—that is, it is a local variable in the `try` block. Local variable type inference with `var` can be used with advantage in declaring resources in this context (§3.13, p. 142). Also, the resource variables are implicitly `final`, guaranteeing that they cannot be assigned to in the `try` block, thus ensuring that the right resources will be closed when the `close()` method is called after the execution of the `try` block. The closing order of the resources is the reverse order in which they are declared in the `try` header—that is, the resource declared first in the `try` header is closed last after the execution of the `try` block. In Example 7.17, the `close()` method is first called on the `BufferedReader` object denoted by the reference `br` as it is declared last, followed by a call to the `close()` method of the `FileReader` object denoted by the reference `fis`. Note how the idempotent behavior of the `close()` method in the `FileReader` object comes into play: It is first called implicitly by the `close()` method of the

`BufferedReader` object, and then again by virtue of the resource declaration statement in the `try` header. The `try`-with-resources statement at (1) can be easily rewritten to avoid this redundant call, where now only the `BufferedReader` resource is declared, ensuring that the `close()` method of the `FileReader` resource will be called only once when the `BufferedReader` resource is closed:

Click here to view code image

```
try (var br = new BufferedReader(new FileReader(args[0]))) {
  // ...
}
```

Example 7.17 *Using* `try` *with Automatic Resource Management (ARM)*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.stream.Stream;

public class TryWithARM {
  public static void main(String[] args) {
    try (var fis = new FileReader(args[0]);        // (1) FileNotFoundExecepti
         var br = new BufferedReader(fis)) {
      String textLine = br.readLine();             // (2) IOException
      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");     // (3) EOFException
      }
    } catch (FileNotFoundException | EOFException e) {
      e.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Running the program:

Click here to view code image

```
>java TryWithARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithARM.main(TryWithARM.java:15)
```

Running the program:

Click here to view code image

```
>java TryWithARM Slogan.txt
Code Compile Compute
```

Concise `try`-with-resources Statement

The header of the `try`-with-resources statement can be made less verbose by factoring out the resource declarations from the header. This refactoring involves *declaring the resources preceding* the `try` statement and specifying instead the *resource variables* in the header. The resources must be `AutoCloseable` (to provide the `close()` method) as before, but in addition must also be *either* `final` *or effectively* `final` to ensure that the right resource is closed.

Click here to view code image

```
final ACResource1 acr1 = new ACResource1();      // (1)
ACResource2 acr2 = new ACResource2();            // (2) Effectively final at (3
// ...
try (acr1; acr2) {                               // (3) Resource variables.
  // Use the resources.
} // Both resources closed on exit.
```

The variable names of the `AutoCloseable` resources declared at (1) and (2) are specified in the header of the `try`-with-resources statement at (3). Apart from the syntactic difference, the semantics of the `try`-with-resources statement are not changed in any way. The resources `acr1` and `acr2` are guaranteed to be closed no matter how the execution of the `try` statement at (3) proceeds.

However, care must be taken if any resource declarations proceeding the `try`-with-resources statement can throw exceptions. For example, if the declaration of `acr2` at (2) above throws an exception, the execution will be interrupted and the resource `acr1` created at (1) will not be closed—leading to resource leakage. One solution is to move the declaration of `acr2` into the header of the `try`-with-resources statement as shown below at (3). The list in the header can be a mix of resource variables and resource declarations. This setup will ensure that both resources will be closed regardless of how the execution in the `try` statement proceeds. The `try`-with-resources statement can be augmented with the necessary `catch` blocks to catch any exceptions thrown during its execution, as shown at (4).

Click here to view code image

```
final ACResource1 acr1 = new ACResource1();                 // (1)
// ...
try (acr1; ACResource2 acr2 = new ACResource2()) {          // (3)
  // Use the resources.
} catch (Exception ex) {                                 // (4)
  ex.printStackTrace();
} // Both resources closed on exit.
```

Implementing the `AutoCloseable` Interface

A declared resource must provide the `close()` method that will be called when the resource is to be closed. This is guaranteed by the fact that the resource must implement the `java.lang.AutoCloseable` interface which specifies the `close()` method. The compiler only allows resource declarations or resource variables in the `try` header that are `AutoCloseable`. Many classes in the Java API implement the `AutoCloseable` interface: byte input streams, byte output streams, readers and writers in the `java.io` package (Chapter 20, p. 1231), and database connections, query statements, and data result sets in the `java.sql` package (Chapter 24, p. 1511).

The abstract `void` method `close()` in the `java.lang.AutoCloseable` interface is declared to throw an `Exception`. Files and streams implement the `java.io.Closeable` interface which extends the `AutoCloseable` interface, but the abstract method `close()` in the `Closeable` interface is specified to throw an `IOException`, a subtype of `Exception`.

Click here to view code image

```
interface AutoCloseable {                           // java.lang package
  void close() throws Exception;
}

interface Closeable extends AutoCloseable {        // java.io package
  void close() throws IOException;                  // Mandatory idempotent
}
```

What is important is that any class implementing the `AutoCloseable` interface can override the `throws` clause of the abstract `close()` method. The implementation of the `close()` method can choose to throw any `Exception`, meaning either an `Exception` or subtypes of `Exception`, but also throw no exception at all if the method cannot fail.

An implementation of the `close()` method of the `Closeable` interface must be idempotent. There is no such requirement for implementations of the `close()` method of the `AutoCloseable` interface, but it is highly recommended that they are idempotent.

In Example 7.18, the class `Gizmo` implements the `AutoCloseable` interface. The class uses the field `closed` to keep track of whether the gizmo is closed or not. The `close()` method at (2) reports whether the gizmo is already closed; if not, it sets the `closed` flag to `true` and throws an unchecked `IllegalArgumentException`. The `close()` method is idempotent, as it will only report that the gizmo is already closed once it has been called. The class `Gizmo` also has the method `compute()` at (3), that throws an unchecked `ArithmeticException`.

The class `GizmoTest` uses the `Gizmo` class. It declares and initializes a `Gizmo` in a `try`-with-resources statement at (4), and calls its `compute()` method. The output shows that the unchecked `ArithmeticException` thrown by the `compute()` method is caught by the `catch` clause at (5), after the `close()` method has been called on the resource—any `catch` clauses and `finally` clause explicitly specified with an extended `try`-with-resources statement are executed *after* the declared resources have been closed.

Example 7.18 *Implementing the* `AutoCloseable` *Interface*

Click here to view code image

```
// File: Gizmo.java
public class Gizmo implements AutoCloseable {
  private boolean closed = false;                // (1) Closed if true
  @Override
  public void close() {                          // (2) Idempotent
    System.out.println("Enter: close()");
    if (closed) {
      System.out.println("Already closed");
    } else {
      closed = true;
      System.out.println("Gizmo closed");
      System.out.println("Throwing IllegalArgumentException in close()");
      throw new IllegalArgumentException("thrown in close()"); // Suppressed
    }
    System.out.println("Exit: close()");     // Only executed if already close
  }

  public void compute() {                         // (3)
    System.out.println("Enter: compute()");
    System.out.println("Throwing ArithmeticException in compute()");
    throw new ArithmeticException("thrown in compute()");
  }
}
```

Click here to view code image

```
// File: GizmoTest.java
public class GizmoTest {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {             // (4)
      myGizmo.compute();
    } catch (Exception ex) {                      // (5)
      System.out.println("Printing stack trace in catch clause of main():");
      ex.printStackTrace();
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

Click here to view code image

```
// File: GizmoTest2.java
public class GizmoTest2 {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {
      myGizmo.compute();
    } catch (Exception ex) {                                        // (6)
      System.out.println("Exception caught in the catch clause of main():\n\t"
                       + ex);
      System.out.println("Printing suppressed exceptions "
                       + "in the catch clause of main():");
      Throwable[] supressedEx = ex.getSuppressed();                 // (7)
      for (Throwable t : supressedEx) {
        System.out.println("\t" + t);
      }
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

Output from running GizmoTest:

Click here to view code image

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Printing stack trace in catch clause of main():
java.lang.ArithmeticException: thrown in compute()
      at Gizmo.compute(Gizmo.java:21)
      at GizmoTest.main(GizmoTest.java:5)
      Suppressed: java.lang.IllegalArgumentException: thrown in close()
          at Gizmo.close(Gizmo.java:13)
          at GizmoTest.main(GizmoTest.java:6)
Finally: Done in main()
```

Output from running GizmoTest2:

Click here to view code image

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Exception caught in the catch clause of main():
      java.lang.ArithmeticException: thrown in compute()
Printing suppressed exceptions in the catch clause of main():
      java.lang.IllegalArgumentException: thrown in close()
Finally: Done in main()
```

Suppressed Exceptions

The program output from `GizmoTest` in Example 7.18 shows that the `Arithmetic-Exception` was thrown in the `compute()` method called in the `try` block at (4) *before* the `IllegalArgumentException` was thrown by the implicit call to the `close()` method. Since only one exception can be propagated, the `IllegalArgumentException` thrown last would mask the `ArithmeticException` and the `IllegalArgumentException` would be propagated. However, the stack trace shows that this is not the case.

Exceptions thrown from the body of a `try`-with-resources statement are given preferential treatment over exceptions thrown by the `close()` method when called implicitly to close a declared resource. Exceptions thrown in the `try` block pertain to the program logic and should not be masked, but at the same time any exceptions thrown in the `close()` method should not be ignored. A solution is provided through *suppressed exceptions*. The class `Throwable` provides the following methods to handle such exceptions:

> Click here to view code image
>
> `void addSuppressed(Throwable exception)`
>
> Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
>
> `Throwable[] getSuppressed()`
>
> Returns an array containing all of the exceptions that were suppressed in order to deliver this exception.

An exception `s` that is associated with an exception `e` through the method call `e.addSuppressed(s)` is called a *suppressed exception*. The idea is that if exception `s` was thrown during the propagation of exception `e`, then exception `s` is suppressed and the propagation of exception `e` continues. When exception `e` is caught, it is possible to retrieve all its suppressed exceptions by the method call `e.getSuppressed()`.

From the stack trace printed by `GizmoTest` in Example 7.18, we see that the `IllegalArgumentException` was thrown in the `close()` method and is implicitly associated with the `ArithmeticException` that was thrown earlier in the `compute()` method—that is, the `IllegalArgumentException` was suppressed and the `ArithmeticException` was propagated.

In the `catch` clause at (6) of `GizmoTest2` in Example 7.18, the exception thrown and the suppressed exception are printed explicitly. The method call `ex.getSuppressed()` at (7) returns an array of `Throwable` containing the suppressed `IllegalArgumentException`. Of course, suppressing any exception from the `close()` method is only warranted if an exception is thrown in the body of the `try`-with-resources statement.

The compiler expands the `try`-with-resources statement into code (essentially a combination of basic `try-catch-finally` blocks with the necessary control flow) that ensures proper closing of the declared resources, and handling of suppressed exceptions.

7.8 Advantages of Exception Handling

Robustness refers to the ability of a software system to respond to errors during execution. A system should respond to unexpected situations at runtime in a responsible way. Applications that provide the user with frequent cryptic messages with error codes or that repeatedly give the user the silent treatment when something goes wrong can hardly be considered robust.

The exception handling mechanism in Java offers the following advantages that facilitate developing robust applications in Java:

- *Separation of exception handling code*

  The code for handling error situations can be separated from the code for the program logic by using the exception handling constructs provided by the language. Code that can result in error situations is confined in the `try` block, and their handling in the `catch` clause.

- *Transparent exception propagation*

  Propagation of a checked exception in the JVM stack cannot be ignored by an active method. The method must comply with the catch-or-declare requirement: Either catch and handle the exception, or propagate it by declaring it in the method's `throws` clause. Error situations causing exception propagation are thus always detected, and can be caught and remedied.

- *Exception categorization and specialization*

  The exception and error classes in the Java SE Platform API are organized in an inheritance hierarchy (Figure 7.3, p. 369). Classes higher in this hierarchy represent *categories* of exceptions and errors (`Exception`, `RuntimeException`, `IOException`, `Error`), whereas classes lower in this hierarchy represent more *specific* exceptions and errors (`NullPointerException`, `FileNotFoundException`, `AssertionError`). The `try-catch` construct allows flexibility in catching and handling exceptions. A `catch` clause can specify an exception category for coarse-grained exception handling, as the exception category class will subsume its more specific exception subclasses, or it can specify a more specific exception class for fine-grained exception handling. Best practice dictates that fine-grained exception handling be used.

Review Questions

7.10 Which of these methods in the class will compile?

Click here to view code image

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ARMy {

  public static void methodA(String filename) throws FileNotFoundExcepti
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodB(String filename)throws IOException {
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodC(String filename)throws FileNotFoundExceptio
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
    catch (IOException ioe) { }
  }

  public static void methodD(String filename) throws FileNotFoundExcepti
    try (var fis = new FileReader(filename);
         var br = new BufferedReader(fis)) { }
  }

  public static void methodE(String filename) throws IOException {
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodF(String filename) {
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodG(String filename)
                                throws IOException {
```

```
      try (FileReader fis = null; BufferedReader br = null) {
        fis = new FileReader(filename);
        br = new BufferedReader(fis);
      }
    }
  }
```

Select the three correct answers.

a. The method at (1).

b. The method at (2).

c. The method at (3).

d. The method at (4).

e. The method at (5).

f. The method at (6).

g. The method at (7).

7.11 What will the following program print when run?

Click here to view code image

```
import java.io.EOFException;
import java.io.IOException;

public class TryTwisting {
  public static void main(String[] args) {
    try {
      justDoIt();
    } catch (Exception t1) {
      System.out.println(t1);
      for (Throwable t : t1.getSuppressed())
        System.out.println("Suppressed: " + t);
    }
  }

  public static void justDoIt() throws Exception {
    IOException t2 = null;
    try {
      t2 = new IOException();
      throw t2;
    } finally {
      try {
        throw new EOFException();
      } catch (Exception t3) {
        t2.addSuppressed(t3);
      }
    }
  }
}
```

Select the one correct answer.

a. `java.io.EOFException`

b. `java.io.IOException`

c. `java.io.Exception`

d. Click here to view code image

```
java.io.EOFException
Suppressed: java.io.IOException
```

e. Click here to view code image

```
java.io.EOFException
Suppressed: java.io.Exception
```

f. Click here to view code image

```
java.io.Exception
Suppressed: java.io.IOException
```

g. Click here to view code image

```
java.io.Exception
Suppressed: java.io.EOFException
```

h. Click here to view code image

```
java.io.IOException
Suppressed: java.io.EOFException
```

i. Click here to view code image

```
java.io.IOException
Suppressed: java.io.Exception
```

7.12 What will make the following program compile and run without throwing an exception?

Click here to view code image

```
public class Truck implements AutoCloseable {
  public void close() {
    throw new Exception("Cannot close.");
  }

  public void load() {
    System.out.println("Loading truck.");
  }

  public static void main(String[] args) {
    try (var tl = new Truck()) {
      tl.load();
    }
  }
}
```

Select the one correct answer.

a. The program will compile and run without reporting any exception.

b. Add a `throws Exception` clause to the `close()` method.

c. Add a `throws Exception` clause to the `main()` method.

d. Add a `throws Exception` clause to both the `close()` and the `main()` methods.

e. Add a `catch (Exception e) {}` clause to the `try` statement in the `main()` method.

f. Add a `throws Exception` clause to the `close()` method and add a `catch (Exception e) {}` clause to the `try` statement in the `main()` method.

g. None of the above

7.13 Which one of these methods will fail to compile? Select the three correct answers.

a. Click here to view code image

```
void task1(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException | FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

b. Click here to view code image

```
void task2(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException e) {
    e = new IOException();
    throw e;
  } catch (FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

c. Click here to view code image

```
void task3() throws IOException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    e = new IOException();
    throw e;
  }
}
```

d. Click here to view code image

```
void task4() throws FileNotFoundException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    throw e;
  }
}
```

e. Click here to view code image

```
void task5() throws Exception {
  try {
    throw new FileNotFoundException();
  } catch (IOException e) {
    throw e;
  }
}
```

7.14 What will be the output of the following program?

Click here to view code image

```
public class Resource implements AutoCloseable {
  public void action() { System.out.print("action "); }
  public void close() throws Exception { System.out.print("closure "); }
}
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (Exception ex) {
      System.out.print("error ");
    }
    System.out.print("the end ");
  }
}
```

Select the one correct answer.

a. action closure the end

b. action the end

c. action error the end

d. `action closure error the end`

7.15 What will be the output of the following program?

Click here to view code image

```
import java.io.IOException;
public class Resource implements AutoCloseable {
  public void action() throws IOException {
    throw new IOException("action error ");
 }
  public void close() throws Exception {
    throw new Exception("closure error ");
 }
}
```

Click here to view code image

```
import java.io.IOException;
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (IOException ex) {
      System.out.print("IO ");
      System.out.print(ex.getMessage());
    } catch (Exception ex) {
      System.out.print("Other ");
      System.out.print(ex.getSuppressed()[0].getMessage());
    }
    System.out.print("the end ");
  }
}
```

Select the one correct answer.

a. `IO action error Other closure error the end`

b. `IO action error the end`

c. `Other closure error IO action error the end`

d. `IO action error`

e. `IO action error Other closure error`

7.16 Which of the following statements is true?

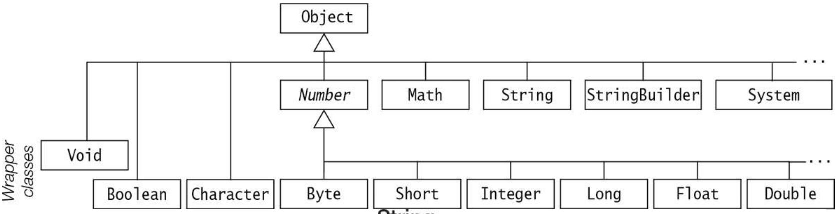
Select the one correct answer.

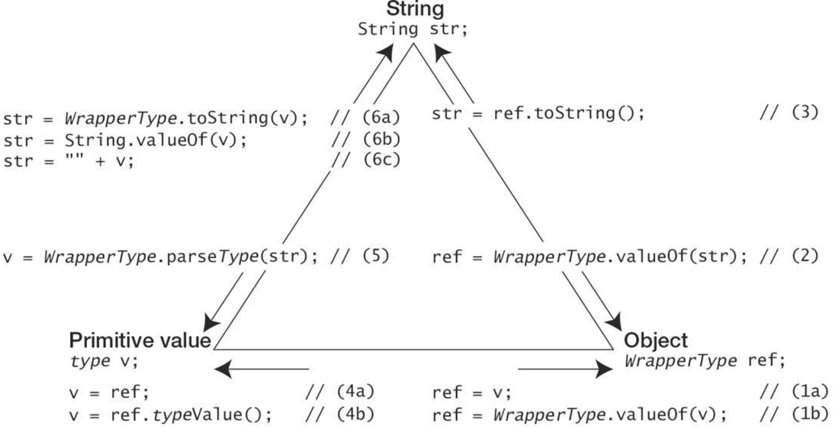
a. Implicit and explicit `finally` blocks cannot coexist with the same `try`-with-resources statement.

b. The implicit `finally` block is executed before any explicit `finally` block.

c. Any explicit `finally` block is executed before the implicit `finally` block.

d. The execution order of the implicit and any explicit `finally` blocks is undetermined.

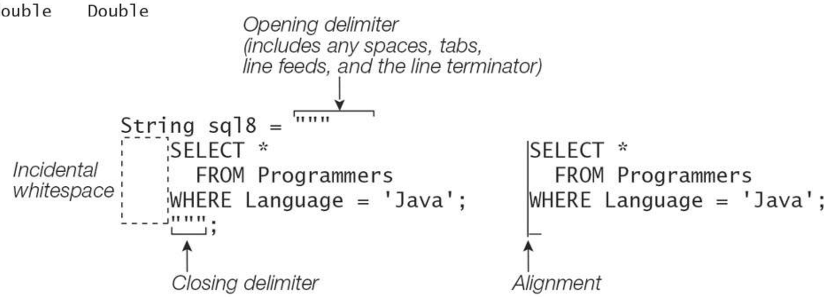

Object
Number
Math
String
StringBuilder
System
Wrapper classes
Void
Boolean
Character
Byte
Short
Integer
Long
Float
Double
...


String
Primitive value
Object

```
String str;

str = WrapperType.toString(v);   // (6a)
str = String.valueOf(v);         // (6b)
str = "" + v;                    // (6c)

str = ref.toString();            // (3)

v = WrapperType.parseType(str); // (5)

ref = WrapperType.valueOf(str); // (2)

type v;

WrapperType ref;

v = ref;                  // (4a)
v = ref.typeValue();      // (4b)

ref = v;                         // (1a)
ref = WrapperType.valueOf(v);    // (1b)
```

`type` is:	*`WrapperType`* is:
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Comments:
(1a) Autoboxing.
(2) Not for `Character` type. Can throw `NumberFormatException`.
(4a) Unboxing.
(5) For numeric wrapper types and `Boolean` only.
Can throw `NumberFormatException` for numeric wrapper types.
(6b) Not for `byte` and `short` primitive types.


Opening delimiter (includes any spaces, tabs, line feeds, and the line terminator)
Incidental whitespace
Closing delimiter
Alignment

```
String sql8 = """
    SELECT *
      FROM Programmers
    WHERE Language = 'Java';
    """;
```

```
SELECT *
  FROM Programmers
WHERE Language = 'Java';
```

(a) Text block *(b) Print out*

```
sql8 ==> "SELECT *\n  FROM Programmers\nWHERE Language = 'Java';\n"
```

(c) The string literal resulting from entering the text block in the `jshell` tool

Selected API Classes 8

Chapter Topics

- An overview of the `java.lang` package
- Understanding the importance of the `Object` class and its API
- Handling primitive values as objects with the wrapper classes
- Converting between primitive values and wrapper classes that represent them, and their text representation
- Creating and manipulating immutable strings with the `String` class, including text blocks
- Creating and manipulating mutable strings with the `StringBuilder` class
- Comparing the `String` and `StringBuilder` classes
- Using common mathematical functions provided by the `Math` class
- Generating pseudorandom numbers using the `Math` and `Random` classes
- Using big numbers for arbitrary-precision arithmetic operations

Java SE 17 Developer Exam O	
[1.1] Use primitives and wrapper classes including Math API, parentheses, type promotion, and casting to evaluate arithmetic and boolean expressions ○ *Only wrapper classes and the Math API are covered in this chapter.* ○ *For primitive data types, operators, conversions, and evaluation of expressions, see Chapter 2, p. 29.*	§8.3, p. 429 §8.6, p. 478
[1.2] Manipulate text, including text blocks, using String and StringBuilder classes	§8.4, p. 439 §8.5, p. 464
Java SE 11 Developer Exam O	
[1.1] Use primitives and wrapper classes, including, operators, the use of parentheses, type promotion and casting ○ *Only wrapper classes are covered in this chapter.*	§8.3, p. 429

For primitive data types, operators, and conversions, see Chapter 2, p. 29.

[1.2] Handle text using String and StringBuilder classes

§8.4, p. 439

§8.5, p. 464

The `Object` class, being the superclass of all classes, defines general functionality for all classes. The wrapper classes provide the support to treat primitive values as objects. The classes `String` and `StringBuilder` provide the support to create and manipulate strings. In addition, the `Math`, `Random`, and `BigDecimal` classes are introduced to leverage mathematical functions, pseudorandom number generators, and arbitrary-precision arithmetic operations, respectively. The APIs of these classes are the main subject of this chapter.

8.1 Overview of the `java.lang` Package

The `java.lang` package is indispensable when programming in Java. It is automatically imported into every source file at compile time. The package contains the `Object` class that is the superclass of all classes, and the wrapper classes (`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) that are used to handle primitive values as objects. It provides classes essential for interacting with the JVM (`Runtime`), for security (`SecurityManager`), for loading classes (`ClassLoader`), for dealing with threads (`Thread`), and for handling exceptions (`Throwable`, `Error`, `Exception`, `Runtime-Exception`). The `java.lang` package also contains classes that provide the standard input, output, and error streams (`System`), string handling (`String`, `StringBuilder`), and mathematical functions (`Math`).

Figure 8.1 shows the important classes whose API is discussed in subsequent sections.

Figure 8.1 *Partial Inheritance Hierarchy in the* `java.lang` *Package*

8.2 The `Object` Class

All classes extend the `Object` class, either directly or indirectly. A class declaration, without the `extends` clause, implicitly extends the `Object` class (§5.1, p. 191). Thus the `Object` class is always at the root of any inheritance hierarchy. The `Object` class defines the basic functionality that all objects exhibit and all classes inherit.

This relationship also applies to arrays, since these are genuine objects in Java.

The `Object` class defines common functionality for all objects through the following methods (see Example 8.1 for usage of these methods):

> Click here to view code image
>
> ```
> public boolean equals(Object obj)
> ```
>
> The general contract of the `equals()` method is discussed in connection with object comparison (§14.2, p. 744). Object reference and value equality are discussed together with the `==` and `!=` operators (§2.13, p. 75). The `equals()` method in the `Object` class returns `true` only if the two references compared denote the same object—that is, they are aliases. The `equals()` method is normally overridden to provide the semantics of object value equality, as is the case for the wrapper classes (p. 432) and the `String` class (p. 439).
>
> ```
> public int hashCode()
> ```
>
> Returns an `int` value as the *hash value* of the object. The general contract of the `hashCode()` method is discussed in connection with object comparison (§14.3, p. 753). In *hash tables*, the hash value can be used to store and retrieve objects efficiently (§15.8, p. 830). The `hashCode()` method is normally overridden by a class, as is the case for the wrapper classes (p. 432) and the `String` class (p. 439).
>
> ```
> public String toString()
> ```
>
> The general contract of the method is to return a text representation of the object. If a subclass does not override this method, it returns a text representation of the object, which has the following format:
>
> ```
> "<class_name>@<hash_code>"
> ```
>
> Since the hash value of an object is an `int` value, this value is printed as a hexadecimal number (e.g., `3e25a5`). This method is usually overridden by a class to provide appropriate text representation for its object. The method call `System.out.println(objRef)` will implicitly convert its argument to a text representation by calling the `toString()` method on the argument.
>
> Click here to view code image

> ```
> public final Class<?> getClass()
> ```
>
> Returns the *runtime class* of the object, which is represented by an object of the class `java.lang.Class` at runtime.
>
> Click here to view code image
>
> ```
> protected Object clone() throws CloneNotSupportedException
> ```
>
> New objects that are exactly the same (i.e., have identical states) as the current object can be created by using the `clone()` method; that is, primitive values and reference values are copied. This is called *shallow copying*. A class can override this method to provide its own notion of cloning. For example, cloning a composite object by recursively cloning the constituent objects is called *deep copying*.
>
> When overridden, the method in the subclass is usually declared as `public` to allow any client to clone objects of the class. Not declaring the overriding method as `public` or `protected` violates the contract for overriding, as it narrows the accessibility of the method, and will be flagged as a compile-time error.
>
> If the overriding `clone()` method in the subclass relies on the `clone()` method in the `Object` class (i.e., a shallow copy), the subclass must implement the `Cloneable` marker interface to indicate that its objects can be safely cloned. Otherwise, the `clone()` method in the `Object` class will throw a checked `CloneNotSupported-Exception`.
>
> Click here to view code image
>
> ```
> @Deprecated(since="9") protected void finalize() throws Throwable
> ```
>
> This method is called on an object just before it is garbage collected so that any clean-up can be done. However, the `finalize()` method in the `Object` class has been deprecated since Java 9—it should no longer be used and it may be removed in the future. The method is deemed problematic, and better solutions exist for implementing the same functionality.

In addition, the following `final` methods in the `Object` class provide support for thread communication in synchronized code (§22.4, p. 1400).

> Click here to view code image

```
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

A thread invokes these methods on the object whose lock it holds. A thread waits for notification by another thread.

Example 8.1 *Methods in the* `Object` *Class*

Click here to view code image

```
// File: ObjectMethods.java
class MyClass implements Cloneable {
  @Override
  public MyClass clone() {
    MyClass obj = null;
    try { obj = (MyClass) super.clone(); }  // Calls overridden method.
    catch (CloneNotSupportedException e) { System.out.println(e);}
    return obj;
  }
}
//______________________________________________________________________________
public class ObjectMethods {
  public static void main(String[] args) {
    // Two objects of MyClass.
    MyClass obj1 = new MyClass();
    MyClass obj2 = new MyClass();

    // Two strings.
    String str1 = new String("WhoAmI");
    String str2 = new String("WhoAmI");

    System.out.println("str1: " + str1.toString());
    System.out.println("str2: " + str2.toString() + "\n");

    // Method hashCode() overridden in String class.
    // Strings that are equal have the same hash code.
    System.out.println("hash code for str1: " + str1.hashCode());
    System.out.println("hash code for str2: " + str2.hashCode() + "\n");

    // Hash codes are different for different MyClass objects.
    System.out.println("hash code for MyClass obj1: " + obj1.hashCode());
    System.out.println("hash code for MyClass obj2: " + obj2.hashCode()+"\n");

    // Method equals() overridden in the String class.
    System.out.println("str1.equals(str2): " + str1.equals(str2));
    System.out.println("str1 == str2:      " + (str1 == str2) + "\n");

    // Method equals() from the Object class called.
    System.out.println("obj1.equals(obj2): " + obj1.equals(obj2));
    System.out.println("obj1 == obj2:      " + (obj1 == obj2) + "\n");

    // The runtime object that represents the class of an object.
    Class<? extends String> rtStringClass  = str1.getClass();
    Class<? extends MyClass> rtMyClassClass = obj1.getClass();
    // The name of the class represented by the runtime object.
    System.out.println("Class for str1: " + rtStringClass);
    System.out.println("Class for obj1: " + rtMyClassClass + "\n");
```

```
      // The toString() method is overridden in the String class.
      String textRepStr = str1.toString();
      String textRepObj = obj1.toString();
      System.out.println("Text representation of str1: " + textRepStr);
      System.out.println("Text representation of obj1: " + textRepObj + "\n");

      // Shallow copying of arrays.
      MyClass[] array1 = {new MyClass(), new MyClass(), new MyClass()};
      MyClass[] array2 = array1.clone();
      // Array objects are different, but share the element objects.
      System.out.println("array1 == array2:        " + (array1 == array2));
      for(int i = 0; i < array1.length; i++) {
        System.out.println("array1[" + i + "] == array2[" + i + "] : " +
                           (array1[i] == array2[i]));
      }
      System.out.println();

      // Clone an object of MyClass.
      MyClass obj3 = obj1.clone();
      System.out.println("hash code for cloned MyClass obj3: " + obj3.hashCode()
      System.out.println("obj1 == obj3: " + (obj1 == obj3));
    }
  }
```

Probable output from the program:

Click here to view code image

```
str1: WhoAmI
str2: WhoAmI

hash code for str1: -1704812257
hash code for str2: -1704812257

hash code for MyClass obj1: 2036368507
hash code for MyClass obj2: 1785210046

str1.equals(str2): true
str1 == str2:      false

obj1.equals(obj2): false
obj1 == obj2:      false

Class for str1: class java.lang.String
Class for obj1: class MyClass

Text representation of str1: WhoAmI
Text representation of obj1: MyClass@7960847b

array1 == array2:        false
array1[0] == array2[0] : true
array1[1] == array2[1] : true
array1[2] == array2[2] : true

hash code for cloned MyClass obj3: 1552787810
obj1 == obj3: false
```

8.3 The Wrapper Classes

Wrapper classes were introduced with the discussion of the primitive data types (Table 2.16, p. 43), and also in connection with boxing and unboxing of primitive values (§2.3, p. 45). Primitive values in Java are not objects. To manipulate these values as objects, the `java.lang` package provides a *wrapper* class for each of the primitive data types (shown in the bottom left of Figure 8.2). The name of the wrapper class is the name of the primitive data type with an uppercase letter, except for `int` (`Integer`) and `char` (`Character`). All wrapper classes are `final`, meaning that they cannot be extended. The objects of all wrapper classes that can be instantiated are *immutable*; in other words, the value in the wrapper object cannot be changed.

Although the `Void` class is considered a wrapper class, it does not wrap any primitive value and is not instantiable (i.e., has no `public` constructors). It just denotes the `Class` object representing the keyword `void`. The `Void` class will not be discussed further in this section.

In addition to the methods defined for constructing and manipulating objects of primitive values, the wrapper classes define useful constants, fields, and conversion methods.

A very important thing to note about the wrapper classes is that their constructors are now *deprecated*. So these constructors should not be used to create wrapper objects, as these constructors will be removed from the API in the future. As we shall see later in this section, the APIs include methods for this purpose that provide significantly improved space and time performance.

Figure 8.2 *Converting Values among Primitive, Wrapper, and* `String` *Types*

Common Wrapper Class Utility Methods

Converting Primitive Values to Wrapper Objects

Autoboxing is a convenient way to wrap a primitive value in an object ((1a) in Figure 8.2 and §2.3, p. 45).

Click here to view code image

```
Character charObj1   = '\n';
Boolean   boolObj1   = true;
Integer   intObj1    = 2020;
Double    doubleObj1 = 3.14;
```

We can also use the following `valueOf(`*type* `v)` method that takes the primitive value of *type* to convert as an argument ((1b) in Figure

8.2).

Click here to view code image

```
static WrapperType valueOf(type v)
```

```
Character charObj1   = Character.valueOf('\n');
Boolean   boolObj1   = Boolean.valueOf(true);
Integer   intObj1    = Integer.valueOf(2020);
Double    doubleObj1 = Double.valueOf(3.14);
```

Converting Strings to Wrapper Objects

Each wrapper class (except `Character`) defines the static method `valueOf(String str)` that returns the wrapper object corresponding to the primitive value represented by the `String` object passed as an argument ((2) in Figure 8.2). This method for the numeric wrapper types also throws a `NumberFormatException` if the `String` parameter is not a valid number.

Click here to view code image

```
static WrapperType valueOf(String str)
```

Click here to view code image

```
Boolean boolObj4   = Boolean.valueOf("false");
Integer intObj3    = Integer.valueOf("1949");
Double  doubleObj2 = Double.valueOf("3.14");
Double  doubleObj3 = Double.valueOf("Infinity");
```

In addition to the one-argument `valueOf()` method, the integer wrapper classes define an overloaded `static valueOf()` method that can take a second argument. This argument specifies the base (or *radix*) in which to interpret the string representing the signed integer in the first argument. Note that the string argument does not specify the prefix for the number system notation.

Click here to view code image

```
static IntegerWrapperType valueOf(String str, int base)
                          throws NumberFormatException
```

Click here to view code image

```
Byte    byteObj1  = Byte.valueOf("1010", 2);     // Decimal value 10
Short   shortObj2 = Short.valueOf("12", 8);      // Decimal value 10
Short   shortObj3 = Short.valueOf("012", 8);     // Decimal value 10
Short   shortObj4 = Short.valueOf("\012", 8);    // NumberFormatException
Integer intObj4   = Integer.valueOf("-a", 16);   // Decimal value -10
Integer intObj6   = Integer.valueOf("-0xa", 16); // NumberFormatException
Long    longObj2  = Long.valueOf("-a", 16);      // Decimal value -10L
```

Converting Wrapper Objects to Strings

Each wrapper class overrides the `toString()` method from the `Object` class. The overriding method returns a `String` object containing the text representation of the primitive value in the wrapper object ((3) in Figure 8.2).

```
String toString()
```

Click here to view code image

```
String charStr   = charObj1.toString();        // "\n"
String boolStr   = boolObj2.toString();        // "true"
String intStr    = intObj1.toString();         // "2020"
String doubleStr = doubleObj1.toString();      // "3.14"
```

Converting Primitive Values to Strings

Each wrapper class defines a `static` method `toString(`*type* `v)` that returns the string corresponding to the primitive value of *type*, which is passed as an argument ((6a) in Figure 8.2).

Click here to view code image

```
static String toString(type v)
```

Click here to view code image

```
String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2020);    // "2020"
String doubleStr2 = Double.toString(3.14);     // "3.14"
```

For integer primitive types, the base is assumed to be 10. For floating-point numbers, the text representation (decimal form or scientific notation) depends on the sign and the magnitude (absolute value) of the number. The NaN value, positive infinity, and negative infinity will result in the strings `"NaN"`, `"Infinity"`, and `"-Infinity"`, respectively.

In addition, the wrapper classes `Integer` and `Long` define methods for converting integers to text representations in decimal, binary, octal, and hexadecimal notation (p. 435).

Converting Wrapper Objects to Primitive Values

Unboxing is a convenient way to unwrap the primitive value in a wrapper object ((4a) in Figure 8.2 and §2.3, p. 45).

Click here to view code image

```
char    c = charObj1;                      // '\n'
boolean b = boolObj2;                      // true
int     i = intObj1;                       // 2020
double  d = doubleObj1;                    // 3.14
```

Each wrapper class defines a *type*`Value()` method that returns the primitive value in the wrapper object ((4b) in Figure 8.2).

```
type typeValue()
```

Click here to view code image

```
char    c = charObj1.charValue();          // '\n'
boolean b = boolObj2.booleanValue();       // true
int     i = intObj1.intValue();            // 2020
double  d = doubleObj1.doubleValue();      // 3.14
```

In addition, each numeric wrapper class defines *type*`Value()` methods for converting the wrapper object to a value of any numeric primitive data type. These methods are discussed later.

Wrapper Comparison, Equality, and Hash Code

Each wrapper class implements the `Comparable<`*Type*`>` interface, which defines the following method:

Click here to view code image

```
int compareTo(Type obj2)
```

This method returns a value that is less than, equal to, or greater than zero, depending on whether the primitive value in the current wrapper *Type* object is less than, equal to, or greater than the primitive value in the wrapper *Type* object denoted by argument `obj2`, respectively.

Click here to view code image

```
// Comparisons based on objects created earlier
Character charObj2   = 'a';
int result1 = charObj1.compareTo(charObj2);      // result1 < 0
int result2 = intObj1.compareTo(intObj3);        // result2 > 0
int result3 = doubleObj1.compareTo(doubleObj2);  // result3 == 0
int result4 = doubleObj1.compareTo(intObj1);     // Compile-time error!
```

Each wrapper class overrides the `equals()` method from the `Object` class. The overriding method compares two wrapper objects for object value equality.

Click here to view code image

```
boolean equals(Object obj2)
```

Click here to view code image

```
// Comparisons based on objects created earlier
boolean charTest   = charObj1.equals(charObj2);      // false
boolean boolTest   = boolObj2.equals(Boolean.FALSE); // false
boolean intTest    = intObj1.equals(intObj3);        // true
boolean doubleTest = doubleObj1.equals(doubleObj2);  // true
boolean test       = intObj1.equals(Long.valueOf(2020L)); // false. Not same t
```

The following values are *interned* when they are wrapped during boxing. That is, only *one* wrapper object exists in the program for these primitive values when boxing is applied:

- The `boolean` value `true` or `false`
- A `byte`
- A `char` with a Unicode value in the interval `[\u0000, \u007f]` (i.e., decimal interval [0, 127])
- An `int` or `short` value in the interval `[-128, 127]`

If references `w1` and `w2` refer to two wrapper objects that box the *same* value, which fulfills any of the conditions mentioned above, then `w1 == w2` is always `true`. In other words, for the values listed previously, object equality and reference equality give the same result.

Click here to view code image

```
// Reference and object equality
Byte bRef1 = 10;
Byte bRef2 = 10;
System.out.println(bRef1 == bRef2);           // true
System.out.println(bRef1.equals(bRef2));      // true

Integer iRef1 = 1000;
Integer iRef2 = 1000;
System.out.println(iRef1 == iRef2);           // false, values not in [-128, 1
System.out.println(iRef1.equals(iRef2));      // true
```

Each wrapper class also overrides the `hashCode()` method in the `Object` class. The overriding method returns a hash value based on the primitive value in the wrapper object.

Click here to view code image

```
int hashCode()
int index = charObj1.hashCode();                     // 10 ('\n')
```

Numeric Wrapper Classes

The numeric wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are all subclasses of the `abstract` class `Number` (Figure 8.1, p. 425).

Each numeric wrapper class defines an assortment of constants, including the minimum and maximum values of the corresponding primitive data type:

Click here to view code image

```
NumericWrapperType.MIN_VALUE
NumericWrapperType.MAX_VALUE
```

The following code retrieves the minimum and maximum values of various numeric types:

Click here to view code image

```
byte   minByte   = Byte.MIN_VALUE;      // -128
int    maxInt    = Integer.MAX_VALUE;   // 2147483647
double maxDouble = Double.MAX_VALUE;    // 1.7976931348623157e+308
```

Converting Numeric Wrapper Objects to Numeric Primitive Types

Each numeric wrapper class defines the following set of *`type`*`Value()` methods for converting the primitive value in the wrapper object to a value of any numeric primitive type:

Click here to view code image

```
byte   byteValue()
short  shortValue()
int    intValue()
long   longValue()
float  floatValue()
double doubleValue()
```

See also (4b) in Figure 8.2.

The following code shows conversion of values in numeric wrapper objects to any numeric primitive type:

Click here to view code image

```
Byte    byteObj2   = (byte)16;                    // Cast mandatory
Integer intObj5    = 42030;
Double  doubleObj4 = Math.PI;
```

```
short  shortVal  = intObj5.shortValue();        // (1)
long   longVal   = byteObj2.longValue();
int    intVal    = doubleObj4.intValue();       // (2) Truncation
double doubleVal = intObj5.doubleValue();
```

Notice the potential for loss of information at (1) and (2), when the primitive value in a wrapper object is converted to a narrower primitive data type.

Converting Strings to Numeric Values

Each numeric wrapper class defines a `static` method `parse`*`Type`*`(String str)`, which returns the primitive numeric value represented by the `String` object passed as an argument. The *Type* in the method name `parse`*Type* stands for the name of a numeric wrapper class, except for the name of the `Integer` class, which is abbreviated to `Int`.

These methods throw a `NumberFormatException` if the `String` parameter is not a valid argument ((5) in Figure 8.2).

Click here to view code image

```
static type parseType(String str) throws NumberFormatException
```

Click here to view code image

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2020");       // parseInt, not parseInteger
int    value3 = Integer.parseInt("7UP");        // NumberFormatException.
double value4 = Double.parseDouble("3.14");
double value5 = Double.parseDouble("Infinity");
```

For the integer wrapper types, the overloaded static method `parse`*`Type`*`()` can additionally take a second argument, which can specify the base in which to interpret the string representing the signed integer in the first argument. Note that the string argument does not specify the prefix for the number system notation.

Click here to view code image

```
type parseType(String str, int base) throws NumberFormatException
```

Click here to view code image

```
byte  value6  = Byte.parseByte("1010", 2);       // Decimal value 10
short value7  = Short.parseShort("12", 8);       // Decimal value 10
short value8  = Short.parseShort("012", 8);      // Decimal value 10
short value9  = Short.parseShort("\012", 8);     // NumberFormatException
int   value10 = Integer.parseInt("-a", 16);      // Decimal value -10
int   value11 = Integer.parseInt("-0xa", 16);    // NumberFormatException
long  value12 = Long.parseLong("-a", 16);        // Decimal value -10L
```

Converting Integer Values to Strings in Different Notations

The wrapper classes `Integer` and `Long` provide static methods for converting integers to text representations in decimal, binary, octal, and hexadecimal notation. Some of these methods from the `Integer` class are listed here, but analogous methods are also defined in the `Long` class. Example 8.2 demonstrates the use of these methods.

> Click here to view code image
>
> ```
> static String toBinaryString(int i)
> static String toHexString(int i)
> static String toOctalString(int i)
> ```
>
> These three methods return a text representation of the integer argument as an *unsigned* integer in base 2, 16, and 8, respectively, with no extra leading zeroes.
>
> Click here to view code image
>
> ```
> static String toString(int i, int base)
> static String toString(int i)
> ```
>
> The first method returns the minus sign (-) as the first character if the integer `i` is negative. In all cases, it returns the text representation of the *magnitude* of the integer `i` in the specified base.
>
> The second method is equivalent to the method `toString(int i, int base)`, where the base has the value 10, and which returns the text representation as a signed decimal ((6a) in Figure 8.2).

Example 8.2 *Text Representation of Integers*

Click here to view code image

```
public class IntegerRepresentation {
  public static void main(String[] args) {
    int positiveInt = +41;    // 0b101001, 051, 0x29
    int negativeInt = -41;    // 0b11111111111111111111111111010111, -0b101001
                              // 037777777727, -051, 0xffffffd7, -0x29
    System.out.println("Text representation for decimal value: " + positiveInt
    integerStringRepresentation(positiveInt);
    System.out.println("Text representation for decimal value: " + negativeInt
    integerStringRepresentation(negativeInt);
  }

  public static void integerStringRepresentation(int i) {
    System.out.println("    Binary:     " + Integer.toBinaryString(i));
```

```
    System.out.println("    Octal:     " + Integer.toOctalString(i));
    System.out.println("    Hex:       " + Integer.toHexString(i));
    System.out.println("    Decimal:   " + Integer.toString(i));

    System.out.println("    Using toString(int i, int base) method:");
    System.out.println("    Base 2:    " + Integer.toString(i, 2));
    System.out.println("    Base 8:    " + Integer.toString(i, 8));
    System.out.println("    Base 16:   " + Integer.toString(i, 16));
    System.out.println("    Base 10:   " + Integer.toString(i, 10));
  }
}
```

Output from the program:

Click here to view code image

```
Text representation for decimal value: 41
    Binary:    101001
    Octal:     51
    Hex:       29
    Decimal:   41
    Using toString(int i, int base) method:
    Base 2:    101001
    Base 8:    51
    Base 16:   29
    Base 10:   41
Text representation for decimal value: -41
    Binary:    11111111111111111111111111010111
    Octal:     37777777727
    Hex:       ffffffd7
    Decimal:   -41
    Using toString(int i, int base) method:
    Base 2:    -101001
    Base 8:    -51
    Base 16:   -29
    Base 10:   -41
```

The `Character` Class

The `Character` class defines a myriad of constants, including the following, which represent the minimum and the maximum values of the `char` type (§2.2, p. 42):

Click here to view code image

```
Character.MIN_VALUE
Character.MAX_VALUE
```

The `Character` class also defines a plethora of `static` methods for handling various attributes of a character, and case issues relating to characters, as defined by the Unicode standard:

Click here to view code image

```
static int     getNumericValue(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
static boolean isTitleCase(char ch)
static boolean isDigit(char ch)
static boolean isLetter(char ch)
static boolean isLetterOrDigit(char ch)
static char    toUpperCase(char ch)
static char    toLowerCase(char ch)
static char    toTitleCase(char ch)
```

The following code converts a lowercase character to an uppercase character:

Click here to view code image

```
char ch = 'a';
if (Character.isLowerCase(ch)) ch = Character.toUpperCase(ch); // A
```

The `Boolean` Class

In addition to the common utility methods for wrapper classes discussed earlier in this section, the `Boolean` class defines the following wrapper objects to represent the primitive values `true` and `false`, respectively:

```
Boolean.TRUE
Boolean.FALSE
```

Converting Strings to Boolean Values

The wrapper class `Boolean` defines the following static method, which returns the `boolean` value `true` only if the `String` argument is equal to the string `"true"`, ignoring the case; otherwise, it returns the `boolean` value `false`. Note that this method does not throw any exceptions, as its numeric counterparts do.

Click here to view code image

```
static boolean parseBoolean(String str)
```

Click here to view code image

```
boolean b1 = Boolean.parseBoolean("TRUE");       // true.
boolean b2 = Boolean.parseBoolean("true");       // true.
boolean b3 = Boolean.parseBoolean("false");      // false.
boolean b4 = Boolean.parseBoolean("FALSE");      // false.
boolean b5 = Boolean.parseBoolean("not true");   // false.
boolean b6 = Boolean.parseBoolean("null");       // false.
boolean b7 = Boolean.parseBoolean(null);         // false.
```

Review Questions

8.1 Which of the following statements is true?

Select the one correct answer.

a. If the references `x` and `y` denote two different objects, the expression `x.equals(y)` is always `false`.
b. If the references `x` and `y` denote two different objects, the expression `(x.hashCode() == y.hashCode())` is always `false`.
c. The `hashCode()` method in the `Object` class is declared as `final`.
d. The `equals()` method in the `Object` class is declared as `final`.
e. All arrays have a method named `clone`.

8.2 What will be the result of compiling and running the following program?

Click here to view code image

```
public class RQ200A70 {
  public static void main(String[] args) {
    Integer i = Integer.valueOf(-10);
    Integer j = Integer.valueOf(-10);
    Integer k = -10;
    System.out.print((i==j) + "|");
    System.out.print(i.equals(j) + "|");
    System.out.print((i==k) + "|");
    System.out.print(i.equals(k));
  }
}
```

Select the one correct answer.

a. `false|true|false|true`
b. `true|true|true|true`
c. `false|true|true|true`
d. `true|true|false|true`
e. None of the above

8.3 Which statement is true about primitives and wrapper classes?

Select the one correct answer.

a. Java automatically converts primitives to wrapper objects with no performance penalty.

b. Java automatically converts wrapper objects to primitives with no performance penalty.

c. Wrapper references can be assigned the `null` value.

d. A variable of a primitive type can be assigned the `null` value.

e. Auto-unboxing of an uninitialized numeric wrapper reference results in a primitive value of zero.

8.4 Given the following code:

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Integer i1 = 10;
    Integer i2 = 10;
    int i3 = 10;
    Integer x1 = i1*i2*i3;
    Integer x2 = i1*i2*i3;
    int x3 = i1*i2*i3;
    String result = (i1 == i2) ? "A" : "";
    result += (i1 == i3) ? "B" : "";
    result += (x1 == x2) ? "C" : "";
    result += (x1 == x3) ? "D" : "";
    System.out.print(result);
  }
}
```

What is the result?

Select the one correct answer.

a. `ABCD`

b. `ABD`

c. `AC`

d. `BD`

e. The program will fail to compile.

8.4 The `String` Class

Handling character sequences is supported primarily by the `String` and `String-Builder` classes. This section discusses the `String` class that provides support for creating, initializing, and manipulating immutable character strings. The next section discusses support for mutable strings provided by the `StringBuilder` class (p. 464).

Internal Representation of Strings

The following character encoding schemes are commonly used for encoding character data on computers:

- LATIN-1: Also known as ISO 8859-1. LATIN-1 is a fixed-length encoding that uses 1 byte to represent a character in the range 00 to FF—that is, characters that can be represented by 8 bits. This encoding suffices for most Western European languages.
- UTF-16: This encoding scheme is a variable-length scheme that uses either 2 bytes or 4 bytes to represent a character in the range 0000 to 10FFFF. This encoding suffices for most languages in the world. However, the `char` type in Java only represents values in the UTF-16 range 0000 to FFFF—that is, characters that can be represented by 2 bytes.

Internally, the character sequence in a `String` object is stored as *an array of* `byte`. If *all* characters in the string can be stored as a *single* byte per character, they are all encoded in the array with the LATIN-1 encoding scheme—1 byte per character. If any character in the sequence requires more than 1 byte, they are all encoded in the array with the UTF-16 encoding scheme—2 bytes per character. To keep track of which encoding is used for the characters in the internal `byte` array, the `String` class has a `private final` encoding-flag field named `coder` which the string methods can consult to correctly interpret the bytes in the internal array. What encoding to use is detected when the string is created. With this strategy of *compact strings*, storage is not wasted as would be the case if all strings were to be encoded in the UTF-16 encoding scheme.

Creating and Initializing Strings

Immutability

The `String` class implements *immutable* character strings, which are read-only once the string has been created and initialized. Objects of the `String` class are thus *thread-safe*, as the state of a `String` object cannot be corrupted through concurrent access by multiple threads. Operations on a `String` object that modify the characters return a new `String` object. The `StringBuilder` class implements mutable strings (p. 464).

String Internment

The easiest way to create a `String` object is to use a string literal:

Click here to view code image

```
String str1 = "You cannot change me!";
```

A string literal is a *reference* to a `String` object. The value in the `String` object is the character sequence that is enclosed in the double quotes of the string literal. Since a string literal is a reference, it can be manipulated like any other `String` reference. The reference value of a string literal can be assigned to another `String` reference: The reference `str1` will denote the `String` object with the value `"You cannot change me!"` after the preceding assignment. A string literal can be used to invoke methods on its `String` object:

Click here to view code image

```
int strLength = "You cannot change me!".length(); // 21
```

The compiler optimizes handling of string literals (and compile-time constant expressions that evaluate to strings): Only one `String` object is shared by all string-valued constant expressions with the same character sequence. Such strings are said to be *interned*, meaning that they share a unique `String` object if they have the same character sequence. The `String` class maintains a private *string pool* where such strings are interned.

Click here to view code image

```
String str2 = "You cannot change me!";      // Interned string assigned.
```

Both `String` references `str1` and `str2` denote the same interned `String` object initialized with the character string: `"You cannot change me!"`. So does the reference `str3` in the following code. The compile-time evaluation of the constant expression involving the two string literals results in a string that is already interned:

Click here to view code image

```
String str3 = "You cannot" + " change me!"; // Compile-time constant expressio
```

In the following code, both the references `can1` and `can2` denote the same interned `String` object, which contains the string `"7Up"`:

Click here to view code image

```
String can1 = 7 + "Up";        // Value of compile-time constant expression: "7
String can2 = "7Up";           // "7Up"
boolean r = can1 == can2;      // true
```

However, in the following code, the reference `can4` denotes a *new* `String` object that will have the value `"7Up"` at runtime:

Click here to view code image

```
String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

The sharing of `String` objects between string-valued constant expressions poses no problem, since the `String` objects are immutable. Any operation performed on one `String` reference will never have any effect on the usage of other references denoting the same object. The `String` class is also declared as `final` so that no subclass can override this behavior. Internally using both compact strings and string internment optimizes memory allocation and performance for strings, which is fully transparent for programmers.

String Constructors

The `String` class has numerous constructors to create and initialize `String` objects based on various types of arguments. Here we present a few selected constructors:

> ```
> String()
> ```
>
> Creates a new `String` object, whose content is the *empty string*, `""`.
>
> ```
> String(String str)
> ```
>
> Creates a new `String` object, whose contents are the same as those of the `String` object passed as the argument.
>
> Click here to view code image
>
> ```
> String(char[] value)
> String(char[] value, int offset, int count)
> ```
>
> Create a new `String` object, whose contents are copied from a `char` array. The second constructor allows extraction of a certain number of characters (`count`) from a given `offset` in the specified array.
>
> Click here to view code image
>
> ```
> String(StringBuilder builder)
> ```
>
> This constructor allows interoperability with the `StringBuilder` class.

Note that using a constructor creates a brand-new `String` object; using a constructor does *not* intern the string. In the following code, the `String` object denoted by `str4` is different from the interned `String` object passed as an argument:

Click here to view code image

```
String str4 = new String("You cannot change me!");
```

Constructing `String` objects can also be done from arrays of bytes, arrays of characters, and string builders:

Click here to view code image

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
StringBuilder strBuilder = new StringBuilder("abba");
//...
String byteStr  = new String(bytes);      // Using array of bytes: "abba"
String charStr  = new String(characters); // Using array of chars: "abba"
String buildStr = new String(strBuilder); // Using string builder: "abba"
```

In Example 8.3, note that the reference `str1` does not denote the same `String` object as the references `str4` and `str5`. Using the `new` operator with a `String` constructor always creates a new `String` object. The expression `"You cannot" + words` is not a constant expression, and therefore, results in the creation of a new `String` object. The local references `str2` and `str3` in the `main()` method and the static reference `str1` in the `Auxiliary` class all denote the same interned string. Object value equality is hardly surprising between these references.

The `String` method `intern()` allows the contents of a `String` object to be interned. If a string with the same contents is not already in the string pool, it is added; otherwise, the already interned string is returned. The following relationship holds between any two strings `strX` and `strY`:

Click here to view code image

```
strX.intern() == strY.intern() is true if and only if strX.equals(strY) is true.
```

In Example 8.3, as the local references `str2` and `str3` in the `main()` method and the static reference `str1` in the `Auxiliary` class all denote the same interned string, the call to the `intern()` method on any of these `String` objects will return the same canonical string from the pool. The method call `str4.intern()` will find that a string with its contents (`"You cannot change me"`) already exists in the string pool, and will also return this string from the pool. Thus the object reference comparison `str1.intern() == str4.intern()` will return `true`, but `str4 == str4.intern()` is still `false`, as the `String` object denoted by `str4` is not interned and therefore does not denote the same `String` object as the one that is interned.

> `String intern()`
>
> If the string pool already contains a string that is equal to this `String` object (determined by the `equals(Object)` method), then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.

Example 8.3 *String Construction and Internment*

Click here to view code image

```
// File: StringConstruction.java
class Auxiliary {
  static String str1 = "You cannot change me!";          // Interned
}
//____________________________________________________________________________
public class StringConstruction {

  static String str1 = "You cannot change me!";          // Interned

  public static void main(String[] args) {
    String emptyStr = new String();                      // ""
    System.out.println("emptyStr: \"" + emptyStr + "\"");

    String str2 = "You cannot change me!";               // Interned
    String str3 = "You cannot" + " change me!";          // Interned
    String str4 = new String("You cannot change me!");   // New String object

    String words = " change me!";
    String str5 = "You cannot" + words;                  // New String object

    System.out.println("str1 == str2:      " +  (str1 == str2));     // (1) tr
    System.out.println("str1.equals(str2): " +  str1.equals(str2));  // (2) tr

    System.out.println("str1 == str3:      " + (str1 == str3));      // (3) tr
    System.out.println("str1.equals(str3): " + str1.equals(str3));   // (4) tr

    System.out.println("str1 == str4:      " + (str1 == str4));      // (5) fa
    System.out.println("str1.equals(str4): " + str1.equals(str4));   // (6) tr

    System.out.println("str1 == str5:      " + (str1 == str5));      // (7) fa
    System.out.println("str1.equals(str5): " + str1.equals(str5));   // (8) tr

    System.out.println("str1 == Auxiliary.str1:      " +
                       (str1 == Auxiliary.str1));        // (9) true
    System.out.println("str1.equals(Auxiliary.str1): " +
                        str1.equals(Auxiliary.str1));    // (10) true

    System.out.println("\"You cannot change me!\".length(): " +
                       "You cannot change me!".length());// (11) 21
    System.out.println("str1.intern() == str4.intern(): " +
                       (str1.intern() == str4.intern()));// (12) true
    System.out.println("str4 == str4.intern(): " +
                       (str4 == str4.intern()));          // (13) false
  }
}
```

Output from the program:

Click here to view code image

```
emptyStr: ""
str1 == str2:       true
str1.equals(str2): true
str1 == str3:       true
str1.equals(str3): true
str1 == str4:       false
str1.equals(str4): true
str1 == str5:       false
str1.equals(str5): true
str1 == Auxiliary.str1:       true
str1.equals(Auxiliary.str1): true
"You cannot change me!".length(): 21
str1.intern() == str4.intern(): true
str4 == str4.intern(): false
```

The `CharSequence` Interface

This interface defines a readable sequence of `char` values. It is implemented by the `String` and `StringBuilder` classes. Many methods in these classes accept arguments of this interface type, and specify it as their return type. This interface facilitates interoperability between these classes. It defines the following methods:

> `int length()`
>
> Returns the number of `char` values in this sequence.
>
> `default boolean isEmpty()`
>
> Returns `true` if this character sequence is empty. The default implementation returns the result of `this.length() == 0`. See also the `String.isBlank()` method (p. 453).
>
> `char charAt(int index)`
>
> A character at a particular index in a sequence can be read using the `charAt()` method. The first character is at index `0` and the last one at index 1 less than the number of characters in the string. If the index value is not valid, an `IndexOutOfBoundsException` is thrown.
>
> Click here to view code image

```
CharSequence subSequence(int start, int end)
```

Returns a new `CharSequence` that is a subsequence of this sequence. Characters from the current sequence are read from the index `start` to the index `end-1`, inclusive.

```
String toString()
```

Returns a string containing the characters in this sequence in the same order as this sequence.

Click here to view code image

```
static int compare(CharSequence cs1, CharSequence cs2)
```

Compares two `CharSequence` instances lexicographically. It returns a negative value, 0, or a positive value if the first sequence is lexicographically less than, equal to, or greater than the second, respectively.

```
default IntStream chars()
```

This default method returns an `IntStream` of `char` values from this sequence (§16.4, p. 901).

Reading Characters from a String

The following methods can be used for character-related operations on a string:

Click here to view code image

`char charAt(int index)` From the `CharSequence` interface `(p. 444).`

```
char[] toCharArray()
```

Returns a new character array, with length equal to the length of this string, that contains the characters in this string.

Click here to view code image

```
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Copies characters from the current string into the destination character array. Characters from the current string are read from index `srcBegin` to the index `srcEnd-1`, inclusive. They are copied into the destination array (`dst`), starting at index `dstBegin` and ending at index `dstbegin+(srcEnd-srcBegin)-1`. The number of characters copied is `(srcEnd-srcBegin)`. An `IndexOutOfBoundsException` is thrown if the indices do not meet the criteria for the operation.

Click here to view code image

`IntStream chars()`	From the `CharSequence` interface `(p. 444).`
`int length()`	From the `CharSequence` interface `(p. 444).`
`boolean isEmpty()`	From the `CharSequence` interface `(p. 444).`

Reading Lines from a String

The following method can be used to extract lines from a string:

> `Stream<String> lines()`
>
> Returns a stream of lines extracted from this string, separated by a *line terminator* (§16.4, p. 902). A line is defined as a sequence of characters terminated by a line terminator.
>
> A line terminator is one of the following: a line feed character `"\n"`, a carriage return character `"\r"`, or a carriage return followed immediately by a line feed `"\r\n"`.

Example 8.4 uses some of these methods for reading characters from strings at (3), (4), (5), and (6). The program prints the frequency of a character in a string and illustrates copying from a string into a character array.

Example 8.4 *Reading Characters from a* `String`

Click here to view code image

```
public class ReadingCharsFromString {
  public static void main(String[] args) {
    int[] frequencyData = new int [Character.MAX_VALUE];    // (1)
    String str = "You cannot change me!";                   // (2)

    // Count the frequency of each character in the string.
    for (int i = 0; i < str.length(); i++) {                // (3)
      try {
        frequencyData[str.charAt(i)]++;                     // (4)
      } catch(StringIndexOutOfBoundsException e) {
        System.out.println("Index error detected: "+ i +" not in range.");
      }
    }

    // Print the character frequency.
    System.out.println("Character frequency for string: \"" + str + "\"");
    for (int i = 0; i < frequencyData.length; i++) {
      if (frequencyData[i] != 0)
        System.out.println((char)i + " (code "+ i +"): " + frequencyData[i]);
    }

    System.out.println("Copying into a char array:");
    char[] destination = new char [str.length() - 3];       // 3 characters le
    str.getChars( 0,            7, destination, 0);         // (5) "You can"
    str.getChars(10, str.length(), destination, 7);         // (6) " change me
                                                            // "not" not copie

    // Print the character array.
    for (int i = 0; i < destination.length; i++) {
      System.out.print(destination[i]);
    }
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Character Frequency for string: "You cannot change me!"
  (code 32): 3
! (code 33): 1
Y (code 89): 1
a (code 97): 2
c (code 99): 2
e (code 101): 2
g (code 103): 1
h (code 104): 1
m (code 109): 1
n (code 110): 3
o (code 111): 2
t (code 116): 1
u (code 117): 1
Copying into a char array:
You can change me!
```

In Example 8.4, the `frequencyData` array at (1) stores the frequency of each character that can occur in a string. The string in question is declared at (2). Since a `char` value is promoted to an `int` value in arithmetic expressions, it can be used as an index in an array. Each element in the `frequencyData` array functions as a frequency counter for the character corresponding to the index value of the element:

Click here to view code image

```
frequencyData[str.charAt(i)]++;                 // (4)
```

The calls to the `getChars()` method at (5) and (6) copy particular substrings from the string into designated places in the `destination` array, before printing the whole character array.

Comparing Strings

Characters are compared based on their Unicode values.

Click here to view code image

```
boolean test = 'a' < 'b';    // true since 0x61 < 0x62
```

Two strings are compared *lexicographically*, as in a dictionary or telephone directory, by successively comparing their corresponding characters at each position in the two strings, starting with the characters in the first position. The string `"abba"` is less than `"aha"`, since the second character `'b'` in the string `"abba"` is less than the second character `'h'` in the string `"aha"`. The characters in the first position in each of these strings are equal.

The following public methods can be used for comparing strings:

> Click here to view code image
>
> ```
> boolean equals(Object obj)
> boolean equalsIgnoreCase(String str2)
> ```
>
> The `String` class overrides the `equals()` method from the `Object` class. The `String` class `equals()` method implements `String` object value equality as two `String` objects having the same sequence of characters. The `equalsIgnoreCase()` method does the same, but ignores the case of the characters.
>
> ```
> int compareTo(String str2)
> ```
>
> The `String` class implements the `Comparable<String>` interface (§14.4, p. 761). The `compareTo()` method compares the two strings, and returns a value based on the outcome of the comparison:
>
> - The value `0`, if this string is equal to the string argument
> - A value less than `0`, if this string is lexicographically less than the string argument
> - A value greater than `0`, if this string is lexicographically greater than the string argument

Here are some examples of string comparisons:

Click here to view code image

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

boolean b1 = strA.equals(strB);               // false
boolean b2 = strA.equalsIgnoreCase(strB);     // true

String str1 = "abba";
String str2 = "aha";

int compVal1 = str1.compareTo(str2);          // negative value => str1 < str2
```

Character Case in a String

> Click here to view code image
>
> ```
> String toUpperCase()
> String toUpperCase(Locale locale)
> String toLowerCase()
> String toLowerCase(Locale locale)
> ```
>
> Note that the original string is returned if none of the characters needs its case changed, but a new `String` object is returned if any of the characters need their case changed. These methods delegate the character-by-character case conversion to corresponding methods from the `Character` class.
>
> These methods use the rules of the (default) *locale* (returned by the method `Locale.getDefault()`, §18.1, p. 1096), which embodies the idiosyncrasies of a specific geographical, political, or cultural region regarding number/date/ currency formats, character classification, alphabet (including case idiosyncrasies), and other localizations.

Examples of case in strings:

Click here to view code image

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");
String strC = strA.toLowerCase();  // Case conversion => New String object:
                                   // "the case was thrown out of court"
String strD = strB.toLowerCase();  // No case conversion => Same String object
String strE = strA.toUpperCase();  // Case conversion => New String object:
                                   // "THE CASE WAS THROWN OUT OF COURT"
boolean test1 = strC == strA;      // false
boolean test2 = strD == strB;      // true
boolean test3 = strE == strA;      // false
```

Concatenation of Strings

Concatenation of two strings results in a new string that consists of the characters of the first string followed by the characters of the second string. The overloaded operator `+` for string concatenation is discussed in §2.8, p. 63. In addition, the following method can be used to concatenate two strings:

```
String concat(String str)
```

The `concat()` method does not modify the `String` object on which it is invoked, as `String` objects are immutable. Instead, the `concat()` method returns a reference to a brand-new `String` object:

Click here to view code image

```
String billboard = "Just";
billboard.concat(" lost in space."); // (1) Returned reference value not store
System.out.println(billboard);       // (2) "Just"
billboard = billboard.concat(" advertise").concat(" here.");  // (3) Chaining.
System.out.println(billboard);       // (4) "Just advertise here."
```

At (1), the reference value of the `String` object returned by the method `concat()` is not stored. This `String` object becomes inaccessible after (1). We see that the reference `billboard` still denotes the string literal `"Just"` at (2).

At (3), two method calls to the `concat()` method are *chained*. The first call returns a reference value to a new `String` object, whose content is `"Just advertise"`. The second method call is invoked on this `String` object using the reference value that was returned in the first method call. The second call results in yet another new `String` object, whose content is `"Just advertise here."` The reference value of this `String` object is assigned to the reference `billboard`. Because `String` objects are immutable, the creation of the temporary `String` object with the content `"Just advertise"` is inevitable at (3).

Some more examples of string concatenation follow:

Click here to view code image

```
String motto = new String("Program once");     // (1)
motto += ", execute everywhere.";              // (2)
motto  = motto.concat(" Don't bet on it!");    // (3)
```

Note that a new `String` object is assigned to the reference `motto` each time in the assignments at (1), (2), and (3). The `String` object with the contents `"Program once"` becomes inaccessible after the assignment at (2). The `String` object with the contents `"Program once, execute everywhere."` becomes inaccessible after (3). The reference `motto` denotes the `String` object with the following contents after execution of the assignment at (3):

Click here to view code image

```
"Program once, execute everywhere. Don't bet on it!"
```

Repeating Strings

The `String` class provides the `repeat()` method that creates a string which is the result of concatenating the current string with itself a specified number of times. The code below illustrates the operation with the `String banner`, which is repeated three times. The method returns a new string, and the original string is intact.

Click here to view code image

```
String banner = "Let's ace that Exam! ";
String bigBanner = banner.repeat(3);
System.out.println("|" + bigBanner + "|");
```

The code prints:

Click here to view code image

```
|Let's ace that Exam! Let's ace that Exam! Let's ace that Exam! |
```

> ```
> String repeat(int count)
> ```
>
> Returns a string whose value is the concatenation of this string repeated `count` number of times. If this string is empty (i.e., its length is zero) then the empty string is returned.

Joining of `CharSequence` Objects

One operation commonly performed on a sequence of strings is to format them so that each string is separated from the next one by a delimiter. For example, given the following sequence of strings:

```
"2014"
"January"
"11"
```

we wish to format them so that individual strings are separated by the delimiter "/":

```
"2014/January/11"
```

The following static methods in the `String` class can be used for this purpose:

> Click here to view code image
>
> ```
> static String join(CharSequence delimiter, CharSequence... elements)
> static String join(CharSequence delimiter,
>                    Iterable<? extends CharSequence> elements)
> ```
>
> Both static methods return a new `String` composed of copies of the `CharSequence` elements joined together with a copy of the specified `CharSequence` delimiter. Thus the resulting string is composed of text representations of the elements separated

> by the text representation of the specified delimiter.
>
> If either `delimiter` is `null` or `elements` is `null`, a `NullPointerException` is thrown. If both are `null`, the method call is ambiguous.
>
> If an element in `elements` is `null`, the string `"null"` is added as its text representation.

Note that both the individual strings and the delimiter string are of type `CharSequence`. The examples in this section use `String` and `StringBuilder` objects that implement the `CharSequence` interface (p. 444).

An `Iterable` provides an iterator to iterate over its elements. The following examples use an `ArrayList` (§12.5, p. 657) that implements the `Iterable` interface. The second `join()` method is then able to iterate over the `Iterable` using the iterator. This method will accept only an `Iterable` whose elements are of type `CharSequence` or are subtypes of `CharSequence`.

The first example shows joining of `String` objects. The first `join()` method is called in this case.

Click here to view code image

```
// (1) Joining individual String objects:
String dateStr = String.join("/", "2014", "January", "11");
System.out.println(dateStr);                          // 2014/January/11
```

The second example shows joining of elements in a `StringBuilder` array. Again the first `join()` method is called, with the array being passed as the second parameter.

Click here to view code image

```
// (2) Joining elements in a StringBuilder array:
StringBuilder left          = new StringBuilder("Left");
StringBuilder right         = new StringBuilder("Right");
StringBuilder[] strBuilders = { left, right, left };
String march = String.join("-->", strBuilders);
System.out.println(march);                            // Left-->Right-->Left
```

The third example shows joining of elements in an `ArrayList` of `StringBuilder`. The second `join()` method is called, with the `ArrayList` being passed as the second parameter. Note that some of the elements of the `ArrayList` are `null`.

Click here to view code image

```
// (3) Joining elements in a StringBuilder list:
ArrayList<StringBuilder> sbList = new ArrayList<>();
```

```
sbList.add(right); sbList.add(null); sbList.add(left); sbList.add(null);
String resultStr = "[" + String.join(", ", sbList) + "]";
System.out.println(resultStr);                    // [Right, null, Left, null]
```

The last example shows joining of elements in an `ArrayList` of `CharSequence`. Again the second `join()` method is called, with the `ArrayList` being passed as the second parameter. Note that elements of the `ArrayList` are `String` and `StringBuilder` objects that are also of type `CharSequence`.

Click here to view code image

```
// (4) Joining elements in a CharSequence list:
ArrayList<CharSequence> charSeqList = new ArrayList<>();
charSeqList.add(right); charSeqList.add(left);     // Add StringBuilder object
charSeqList.add("Right"); charSeqList.add("Left"); // Add String objects.
String resultStr2 = "<" + String.join("; ", charSeqList) + ">";
System.out.println(resultStr2);                   // <Right; Left; Right; Left>
```

Searching for Characters and Substrings in Strings

The following overloaded methods can be used to find the index of a character or the start index of a substring in a string. These methods search *forward* toward the end of the string. In other words, the index of the *first* occurrence of the character or substring is found. If the search is unsuccessful, the value `-1` is returned.

Click here to view code image

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
```

The first method finds the index of the first occurrence of the argument character in a string. The second method finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument. If the index argument is negative, the index is assumed to be 0. If the index argument is greater than the length of the string, it is effectively considered to be equal to the length of the string, resulting in the value `-1` being returned.

Click here to view code image

```
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

The first method finds the start index of the first occurrence of the substring argument in a string. The second method finds the start index of the first occurrence of the substring

> argument in a string, starting at the index specified in the second argument.

The `String` class also defines a set of methods that search for a character or a substring, but the search is *backward* toward the start of the string. In other words, the index of the *last* occurrence of the character or substring is found.

Click here to view code image

```
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

The following methods can be used to create a string in which all occurrences of a character or a subsequence in a string have been replaced with another character or subsequence:

> Click here to view code image
>
> ```
> String replace(char oldChar, char newChar)
> String replace(CharSequence target, CharSequence replacement)
> ```
>
> The first method returns a new `String` object that is the result of replacing all occurrences of the `oldChar` in the current string with the `newChar`. The current string is returned if no occurrences of the `oldChar` can be found.
>
> The second method returns a new `String` object that is the result of replacing all occurrences of the character sequence `target` in the current string with the character sequence `replacement`. The current string is returned if no occurrences of the `target` can be found.

The following methods can be used to test whether a string satisfies a given criterion:

> Click here to view code image
>
> ```
> boolean contains(CharSequence cs)
> ```
>
> Returns `true` if the current string contains the specified character sequence, and `false` otherwise.
>
> Click here to view code image
>
> ```
> boolean startsWith(String prefix)
> ```
>
> Returns `true` if the current string starts with the character sequence specified by parameter `prefix`, and `false`

otherwise.

Click here to view code image

```
boolean startsWith(String prefix, int index)
```

Returns `true` if the substring of the current string at the specified `index` starts with the character sequence specified by parameter `prefix`, and `false` otherwise.

Click here to view code image

```
boolean endsWith(String suffix)
```

Returns `true` if the current string ends with the character sequence specified by parameter `suffix`, and `false` otherwise.

Examples of search and replace methods in the `String` class:

Click here to view code image

```
String funStr = "Java Jives";
//               0123456789

int jInd1a = funStr.indexOf('J');             // 0
int jInd1b = funStr.indexOf('J', 1);          // 5
int jInd2a = funStr.lastIndexOf('J');         // 5
int jInd2b = funStr.lastIndexOf('J', 4);      // 0

String banner = "One man, One vote";
//               01234567890123456

int subInd1a = banner.indexOf("One");         // 0
int subInd1b = banner.indexOf("One", 3);      // 9
int subInd2a = banner.lastIndexOf("One");     // 9
int subInd2b = banner.lastIndexOf("One", 10); // 9
int subInd2c = banner.lastIndexOf("One", 8);  // 0
int subInd2d = banner.lastIndexOf("One", 2);  // 0

String newStr    = funStr.replace('J', 'W');      // "Wava Wives"
String newBanner = banner.replace("One", "No");   // "No man, No vote"
boolean found1   = banner.contains("One");        // true
boolean found2   = newBanner.contains("One");     // false

String song = "Start me up!";
//             012345677890
boolean found3    = song.startsWith("Start");     // true
boolean notFound1 = song.startsWith("start");     // false
boolean found4    = song.startsWith("me", 6);     // true
boolean found5    = song.endsWith("up!");         // true
boolean notFound2 = song.endsWith("up");          // false
```

Extracting Substrings from Strings

The `String` class provides methods to trim and strip strings, and also extract substrings.

```
boolean isBlank()
```

Returns `true` if the string is empty or contains only *whitespace*; otherwise, it returns `false`. See also the method `isEmpty()` in the `CharSequence` interface (p. 444).

```
String strip()
String stripLeading()
String stripTrailing()
```

Return a string whose value is this string, with all leading and trailing *whitespace* removed, or with all leading whitespace removed, or with all trailing whitespace removed, respectively. If this `String` object represents an empty string, or if all characters in this string are whitespace, then an empty string is returned. See also the method `stripIndent()` (p. 444).

```
String trim()
```

This method can be used to create a string where all characters with values less than or equal to the space character `'\u0020'` have been removed from the front (leading) and the end (trailing) of a string. It is recommended to use the strip methods to remove leading and trailing whitespace in a string.

Click here to view code image

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

The `String` class provides these overloaded methods to extract substrings from a string. A new `String` object containing the substring is created and returned. The first method extracts the string that starts at the given index `startIndex` and extends to the end of the string. The end of the substring can be specified by using a second argument `endIndex` that is the index of the first character *after* the substring—that is, the last character in the substring is at index `endIndex-1`. If the index value is not valid, an `IndexOutOfBoundsException` is thrown.

Click here to view code image

`CharSequence subSequence(int start, int end)` From the `CharSequence` interface (p. 444)

This method behaves the same way as the `substring(int start, int end)` method.

The `Character.isWhitespace()` method can be used to determine whether a character is whitespace.

Click here to view code image

```
System.out.println(Character.isWhitespace('\t'));  // true
System.out.println(Character.isWhitespace('\n'));  // true
System.out.println(Character.isWhitespace('a'));   // false
```

Examples of blank strings:

Click here to view code image

```
System.out.println("".isBlank());        // true
System.out.println(" \t  ".isBlank());   // true
```

Examples of stripping a string:

Click here to view code image

```
String utopia = "\t\n  Java Nation \n\t  ";
System.out.println(utopia.strip().equals("Java Nation"));               // tru
System.out.println(utopia.stripLeading().equals("Java Nation \n\t  ")); // tru
System.out.println(utopia.stripTrailing().equals("\t\n  Java Nation")); // tru
```

Examples of extracting substrings:

Click here to view code image

```
String utopia2 = "\t\n  Java Nation \n\t  ";
utopia2 = utopia2.trim();                       // "Java Nation"
utopia2 = utopia2.substring(5);                 // "Nation"
String radioactive = utopia2.substring(3,6);    // "ion"
```

Converting Primitive Values and Objects to Strings

The `String` class overrides the `toString()` method in the `Object` class and returns the `String` object itself:

Click here to view code image

`String toString()` From the `CharSequence` interface (p. 444).

The `String` class also defines a set of static overloaded `valueOf()` methods to convert objects and primitive values into strings:

Click here to view code image

```
static String valueOf(Object obj)
static String valueOf(char[] charArray)
static String valueOf(boolean b)
static String valueOf(char c)
```

> All of these methods return a string representing the given parameter value. A call to the method with the parameter `obj` is equivalent to `obj.toString()` when `obj` is not `null`; otherwise, the `"null"` string is returned. The `boolean` values `true` and `false` are converted into the strings `"true"` and `"false"`. The `char` parameter is converted to a string consisting of a single character.
>
> Click here to view code image
>
> ```
> static String valueOf(int i)
> static String valueOf(long l)
> static String valueOf(float f)
> static String valueOf(double d)
> ```
>
> The static `valueOf()` method, which accepts a primitive value as an argument, is equivalent to the static `toString()` method in the corresponding wrapper class for each of the primitive data types ((6a) and (6b) in §8.3, p. 430). Note that there are no `valueOf()` methods that accept a `byte` or a `short`.

Examples of string conversions:

Click here to view code image

```
  String anonStr    = String.valueOf("Make me a string.");       // "Make me a str
  String charStr    = String.valueOf(new char[] {'a', 'h', 'a'});// "aha"
  String boolTrue   = String.valueOf(true);                        // "true"
  String doubleStr  = String.valueOf(Math.PI);                     // "3.1415926535
```

Transforming a String

The `transform()` method allows a function to be applied to a string to compute a result. The built-in functional interface `Function<T, R>`, and writing lambda expressions to implement such functions, are covered in the following sections: §13.8, p. 712, and §13.2, p. 679, respectively.

> Click here to view code image
>
> ```
> <R> R transform(Function<? super String,? extends R> f)
> ```
>
> The specified function `f` is applied to this string to produce a result of type `R`.

The example below illustrates using the `transform()` method to reverse the characters in a string via a `StringBuilder` (p. 464). Its versatility becomes apparent when the string contains lines of text or a text block that needs to be processed (p. 458).

Click here to view code image

```
String message = "Take me to your leader!";
String tongueSpeake = message.transform(s ->
    new StringBuilder(s).reverse().toString().toUpperCase());
System.out.println(tongueSpeake); // !REDAEL RUOY OT EM EKAT
```

Indenting Lines in a String

The `indent()` method allows indentation of lines in a string to be adjusted.

> `String indent(int n)`
>
> Adjusts the indentation of each line of this string based on the value of `n`, and normalizes line termination characters.
>
> This string is conceptually separated into lines using the `String.lines()` method (p. 445). Each line is then adjusted depending on the value of `n`, and then terminated with a line feed (`"\n"`). The resulting lines are then concatenated and the final string returned.
>
> If `n > 0` then `n` spaces are inserted at the beginning of each line. This also applies to an empty line.
>
> If `n == 0` then the line remains unchanged. However, line terminators are still normalized.
>
> If `n < 0` then at most `n` whitespace characters are removed from the beginning of each line, depending on the number of leading whitespace characters in the line. Each whitespace character is treated as a single character; specifically, the tab character `"\t"` is considered a single character and is not expanded.

The example below illustrates how indentation is adjusted depending on the value passed to the `indent()` method. The string `cmds` constructed at (0) has four lines, where the third line is an empty line. The first, second, and last lines are indented by one, two, and three spaces, respectively. The first line is terminated with the return character (`\r`), and the last line is not terminated at all. The other lines are terminated normally with a newline character (`\n`). Keep also in mind that the string `cmds` is immutable.

Click here to view code image

```
String cmds = " Attention!\r  Quick march!\n\n   Eyes left!"; // (0)
String str1 = cmds.indent(0);                                 // (1)
String str2 = cmds.indent(3);                                  // (2)
String str3 = cmds.indent(-1);                                 // (3)
String str4 = cmds.indent(-2);                                 // (4)
String str5 = cmds.indent(-3);                                 // (5)
```

The lines numbered (1) to (5) below show the resulting string from the method calls above from (1) to (5), respectively, applied to the string `cmds` at (0) above. The `jshell` tool can readily be used to execute the method calls above to examine the resulting strings. Note that termination of all lines is normalized with the newline character (`\n`).

Click here to view code image

```
(0) cmds ==> " Attention!\r  Quick march!\n\n   Eyes left!"
(1) str1 ==> " Attention!\n  Quick march!\n\n   Eyes left!\n"
(2) str2 ==> "    Attention!\n     Quick march!\n   \n      Eyes left!\n"
(3) str3 ==> "Attention!\n Quick march!\n\n  Eyes left!\n"
(4) str4 ==> "Attention!\nQuick march!\n\n Eyes left!\n"
(5) str5 ==> "Attention!\nQuick march!\n\nEyes left!\n"
```

In Table 8.1, the columns correspond to the resulting strings from (1) to (5) above. The last row in Table 8.1 shows the output when the resulting strings are printed.

The underscore (_) in each cell in the last row is a visual marker that indicates the position after printing the resulting string. Note that the empty line at (0) above is indented (marked visually as *sss*), as shown in column (2). A value less than `-3` does not change the result shown in column (5).

Table 8.1 *Indenting Lines in a String*

123456789		(5) n = -3		
Attention! Quick march! *sss* Eyes left! _				

Formatted Strings

We have used the `System.out.printf()` method to format values and print them to the terminal window (§1.9, p. 24). To just create the string with the formatted values, but not print the formatted result, we can use the following static method from the `String` class. It accepts the same arguments as the `printf()` method, and uses the same format specifications (Table 1.2, p. 26).

> Click here to view code image
>
> ```
> static String format(String format, Object... args)
> ```
>
> Returns a string with the result of formatting the values in the variable arity parameter `args` according to the `String` parameter `format`. The format string contains format specifications that determine how each subsequent value in the variable arity parameter `args` will be formatted.
>
> Any error in the format string will result in a runtime exception.
>
> Click here to view code image
>
> ```
> String formatted(Object... args)
> ```
>
> Formats the supplied arguments using this string as the format string. It is equivalent to `String.format(this, args)`.

The following call to the `format()` method creates a formatted string with the three values formatted according to the specified format string:

Click here to view code image

```
String formattedStr = String.format("Formatted values|%5d|%8.3f|%5s|",
                                    2020, Math.PI, "Hi");
System.out.println(formattedStr);  // Formatted values| 2020|   3.142|   Hi|
formattedStr = formattedStr.toUpperCase();
System.out.println(formattedStr);  // FORMATTED VALUES| 2020|   3.142|   HI|
```

Alternatively, we can use the `formatted()` method:

Click here to view code image

```
String formattedStr1 = "Formatted values|%5d|%8.3f|%5s|"
                          .formatted(2020, Math.PI, "Hi");
System.out.println(formattedStr1); // Formatted values| 2020|   3.142|   Hi|
```

For formatting strings in a language-neutral way, the `MessageFormat` class can be considered (§18.7, p. 1139). Other miscellaneous methods exist in the `String` class for pattern matching (`matches()`), splitting strings (`split()`), and converting a string to an array of bytes (`getBytes()`). The method `hashCode()` can be used to compute a hash value based on the characters in the string. Please consult the Java SE Platform API documentation for more details.

Text Blocks

Constructing string literals that span multiple lines can be tedious using string concatenation and line terminators. Text blocks provide a better solution—avoiding having to escape tab, newline, and double-quote characters in the text, and in addition preserving indentation of a multiline string.

Basic Text Blocks

The string `sql1` below represents a three-line SQL query. It is constructed using a *text block*, resulting in an object of type `String`. We will use it as an example to illustrate constructing text blocks.

Click here to view code image

```
String sql1 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql1 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n";
```

Printing the text block `sql1` above will give the following result, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers
WHERE Language = 'Java';
_
```

A text block starts with three double quotes (`"""`) and ends with three double quotes (`"""`). The *opening delimiter* of a text block consists of the opening three double quotes (followed by any spaces, tabs, or form feed characters) that must be terminated by a line terminator. The *closing delimiter* of a text block consists of three double quotes.

The *content of a text block* begins with the sequence of characters that immediately *begins after the line terminator* of the opening delimiter and *ends with the first double quote* of the closing delimiter. Note that the line terminator of the opening delimiter is *not* part of the text block's content.

The following attempt to construct a text block will result in a compile-time error because of erroneous characters (i.e., the `//` comment) that follow the opening three double quotes of the text block.

Click here to view code image

```
String badBlock = """      // (1) Compile-time error!
Not a good start.
""";
```

The text block below results in an empty string (`""`):

Click here to view code image

```
String emptyBlock = """
""";
```

Using Escape Sequences in Text Blocks

We do not need to end each text line with the `\n` escape sequence, as the text block retains the line terminators entered directly into the text. However, using the `\n` escape sequence will be interpreted as literally inserting a newline character in the text. The query `sql2` below is equivalent to the query `sql1` above, and will result in the same string. (For escape sequences, see Table 2.10, p. 38.)

Click here to view code image

```
String sql2 = """
SELECT *
FROM Programmers\nWHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql2 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n"
```

However, ending a line explicitly in the text block with a newline character will result in an empty line being inserted into the resulting string, as can be seen in the code below:

Click here to view code image

```
String sql3 = """
SELECT *
FROM Programmers\n
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql3 ==> "SELECT *\nFROM Programmers\n\nWHERE Language = 'Java';\n"
```

Printing the text block `sql3` above will give the following result, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers

WHERE Language = 'Java';
_
```

In the examples so far, the closing delimiter of a text block was specified on the last line by itself, resulting in the *last line of text* in the text block being terminated by the line terminator that was entered directly. If the last line of text should not be terminated, the closing delimiter can be used at the end of this line as shown below.

Click here to view code image

```
String sql4 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';"""; // No line terminator. Closing delimiter ends blo
```

The string literal resulting from entering the text block in the `jshell` tool shows no line terminator for the last line of the query:

Click here to view code image

```
sql4 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';"
```

The print statement below does not print a newline after the last line:

Click here to view code image

```
System.out.print(sql4);
SELECT *
FROM Programmers
WHERE Language = 'Java';
```

Other escape sequences (Table 2.10, p. 38) can also be used in a text block. In contrast to the `\n` escape sequence, the *\Line terminator* escape sequence, allowed only in a text block, escapes the line terminator, thus preventing the termination of the current line so that it is joined with the next line.

Click here to view code image

```
String sql5 = """
SELECT * \
FROM Programmers \
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool shows that the three lines were joined and no characters were substituted for the *Line terminator* escape sequence:

Click here to view code image

```
sql5 ==> "SELECT * FROM Programmers WHERE Language = 'Java';\n"
```

The result from printing the string `sql5` is shown below, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string.

Click here to view code image

```
SELECT * FROM Programmers WHERE Language = 'Java';
_
```

To include a backslash (`\`) in any context other than line continuation will require escaping the backslash, analogous to using it in a string literal.

By default, any trailing whitespace on a line is removed and line termination normalized. In some cases, it might be necessary to retain trailing whitespace. This can be achieved by using the `\s` escape sequence. Replacing the last trailing space that should be retained with this escape sequence will preserve any trailing whitespace *before* the `\s` escape sequence, including the space replaced by the escape sequence. At (1) below, *four* trailing spaces are retained.

```
String sql6 = """
SELECT *
```

Click here to view code image

```
FROM Programmers   \s     """;                   // (1) No line termination.
// sql6 ==> "SELECT *\nFROM Programmers    "

String sql7 = "WHERE Language = 'Java';\n";
System.out.print(sql6 + sql7);
```

Output from the code, showing the retained spaces, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers    WHERE Language = 'Java';
_
```

Double quotes (") are treated like any other character and can be used without escaping them in a text block, except when a three double quote sequence is used in the text. It is enough to escape the first double quote in a three double quote sequence, but all three double quotes can also be escaped individually in the sequence. It is also not required that double quotes should be balanced. The code below illustrates using double quotes in a text block.

Click here to view code image

```
String fact = """
The sequence \"""
has "special" meaning in a text block.
""";
// fact ==> "The sequence \"\"\"\nhas \"special\" meaning in a text block.\n"
System.out.print(fact);
```

Output from the code, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
The sequence """
has "special" meaning in a text block.
_
```

A text block can be used to define the *format string* for value substitution in formatted text (p. 457). Below, the text block defines the format string used by the `formatted()` method for substituting values of its arguments.

Click here to view code image

```
String query = """
SELECT *
FROM %s
WHERE %s = '%s';
""".formatted("Customers", "Country", "Norway");
System.out.print(query);
```

Output from the print statement, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Customers
WHERE Country = 'Norway';
_
```

Incidental Whitespace

So far the lines in the text blocks have all been left-justified. This need not be the case. Indentation can be used if desired. However, one needs to be careful as *incidental whitespace* from *each* line is removed by the compiler. Incidental whitespace is indentation that is common to all lines in the text block. It is equal to the number of leading whitespace characters in the *least indented* lines in the text block. The least indented lines end up with no indentation. The line containing the closing delimiter also plays a part in determining the incidental whitespace to be removed from all lines. Regardless of how much incidental whitespace is removed from the other lines in the text box, *all* leading whitespace preceding the closing delimiter is always removed, resulting in an empty line.

Whitespace that is not incidental is referred to as *essential whitespace*.

Figure 8.3 *Incidental Whitespace in Text Blocks*

In Figure 8.3a, we see that the first and third lines, and the line with the closing delimiter, are the ones that are least indented in the text block. The number of leading whitespace characters in each of these lines is four. The indentation in the second line is six positions. The least number (i.e., four) of leading whitespace characters common to all lines is the incidental whitespace that is removed from *each* line. Figure 8.3b shows how the contents of the text block will be printed. The underscore (_) is a visual marker that indicates the position after printing the text block, showing that a newline is printed for the last line of the text block. Removing the incidental whitespace preserves the relative indentation of the lines in the text block. Figure 8.3c shows the string literal resulting from entering the text block in the `jshell` tool.

As stated earlier, the whitespace preceding the closing delimiter is significant in determining incidental whitespace. Table 8.2 shows a few more examples to illustrate how incidental whitespace is calculated and removed in a text block. The last two rows show the text block declaration and the result of printing the text block, respectively. The underscore (_) in each cell in the last row is a visual marker that indicates the position after printing the text block, showing that no newline is printed for the last line of the text blocks in columns (4) and (5). The numbered remarks below refer to the column numbers in Table 8.2. The string literal resulting from entering each text block in the `jshell` tool is also shown.

(1) The least indentation (0) is determined by the last line with the closing delimiter. No incidental whitespace is removed.

Click here to view code image

```
tb1 ==> "    KEEP\n      IT\n        SIMPLE\n"
```

(2) The least indentation (2) is determined by the last line with the closing delimiter.

Click here to view code image

```
tb2 ==> "  KEEP\n    IT\n      SIMPLE\n"
```

(3) The least indentation (4) is determined by the line containing the word `IT`. However, *all* leading whitespace in the *last* line is removed, not just four spaces as in the other lines—we can see the result in the string literal below:

Click here to view code image

```
tb3 ==> "  KEEP\nIT\n  SIMPLE\n"
```

(4) The least indentation (2) is determined by the last line containing the word `SIMPLE` and the closing delimiter. Note that the last line is not terminated.

Click here to view code image

```
tb4 ==> "  KEEP\n    IT\nSIMPLE"
```

(5) The least indentation (4) is determined by the lines containing the words `KEEP` and `IT`, respectively. Note that in the last line only incidental whitespace is removed, and the line is not terminated.

Click here to view code image

```
tb5 ==> "KEEP\nIT\n  SIMPLE"
```

Table 8.2 *Incidental Whitespace in Text Blocks*

Least indent is 2 123456789012345		**(5)**		
String tb4 = """ KEEP IT SIMPLE"; """; """;				
KEEP IT SIMPLE =				

A blank line (i.e., one that is empty or contains only whitespace) in a text block is replaced by an empty line—that is, any whitespace it contains is removed.

Incidental whitespace in a text block is processed internally as if by the execution of the `String.stripIndent()` method. Calling this method on a text block with one or more lines has *no* effect on the content of the text block, but it will strip any whitespace from a text block that has *no line structure*.

The reader is encouraged to use the `jshell` tool to examine the string literal resulting from entering the text blocks presented in this section.

> `String stripIndent()`
>
> Returns a string whose value is this string, with incidental whitespace removed from the beginning and end of every line.

8.5 The `StringBuilder` Class

Although there is a close relationship between objects of the `String` and `StringBuilder` classes, these are two independent `final` classes, both directly extending the `Object` class. Hence, `String` references cannot be stored (or cast) to `StringBuilder` references, and vice versa. However, both classes implement the `CharSequence` interface (p. 444) and the `Comparable` interface (§14.4, p. 761).

Since the `StringBuilder` class does not override the `equals()` and `hashCode()` methods from the `Object` class, the contents of string builders should be converted to `String` objects for equality comparison and to compute a hash value.

The `StringBuilder` class has many operations analogous to the ones in the `String` class. In addition, it provides support for manipulating *mutable* strings (p. 467).

Constructing String Builders

Mutability

In contrast to the `String` class, which implements immutable character sequences, the `StringBuilder` class implements *mutable* character sequences. Not only can the character sequences in a string builder be changed, but the capacity of the string builder can also change dynamically. The *capacity* of a string builder is the maximum number of characters that a string builder can accommodate before its size is automatically augmented.

The legacy class `StringBuffer` (Figure 8.1, p. 425) also implements *mutable* sequences of characters. It support the same operations as the `StringBuilder` class, but the `StringBuffer` class is *thread-safe*. Certain operations on a string buffer are synchronized so that when accessed concurrently by multiple threads, these operations are safe to perform without corrupting the state of the string buffer (§22.4, p. 1387). Note that a `String` object is also thread-safe—because it is immutable, a thread cannot change its state. String builders are preferred when heavy modification of character sequences is involved and synchronization of operations, which also carries a performance penalty, is not important. Although the rest of this section focuses on string builders, it is equally applicable to string buffers.

String Builder Constructors

The `final` class `StringBuilder` provides four constructors that create and initialize `StringBuilder` objects and set their initial capacity.

> Click here to view code image
>
> ```
> StringBuilder(String str)
> StringBuilder(CharSequence charSeq)
> ```
>
> The contents of the new `StringBuilder` object are the same as the contents of the `String` object or the character sequence passed as an argument. The initial capacity of the string builder is set to the length of the argument sequence, plus room for 16 more characters.
>
> Click here to view code image

> `StringBuilder(int initialCapacity)`
>
> The new `StringBuilder` object has no content. The initial capacity of the string builder is set to the value of the argument, which cannot be less than 0.
>
> `StringBuilder()`
>
> This constructor also creates a new `StringBuilder` object with no content. The initial capacity of the string builder is set to 16 characters.

Examples of `StringBuilder` object creation and initialization:

Click here to view code image

```
StringBuilder strBuilder1 = new StringBuilder("Phew!");  // "Phew!", capacity
StringBuilder strBuilder2 = new StringBuilder(10);       // "", capacity 10
StringBuilder strBuilder3 = new StringBuilder();         // "", capacity 16
```

Reading Characters from String Builders

The following methods can be used for reading characters in a string builder:

> Click here to view code image
>
> `char charAt(int index)`   From the `CharSequence` interface (p. 444).
>
> `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
>
> Copies characters from the current string builder into the destination character array. Characters from the current string builder are read from index `srcBegin` to index `srcEnd-1`, inclusive. They are copied into the destination array (`dst`), starting at index `dstBegin` and ending at index `dstbegin+(srcEnd-srcBegin)-1`. The number of characters copied is `(srcEnd-srcBegin)`. An `IndexOutOfBoundsException` is thrown if the indices do not meet the criteria for the operation.
>
> Click here to view code image
>
> `IntStream chars()`   From the `CharSequence` interface (p. 444).
>
> `int length()`   From the `CharSequence` interface (p. 444).

The following is an example of reading the contents of a string builder:

Click here to view code image

```
StringBuilder strBuilder = new StringBuilder("Java");      // "Java", capacity
char charFirst = strBuilder.charAt(0);                     // 'J'
char charLast  = strBuilder.charAt(strBuilder.length()-1); // 'a'
```

Searching for Substrings in String Builders

The methods for searching substrings in string builders are analogous to the ones in the `String` class (p. 451).

Click here to view code image

```
int indexOf(String substr)
int indexOf(String substr, int fromIndex)

int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

Examples of searching for substrings in a `StringBuilder`:

Click here to view code image

```
StringBuilder banner2 = new StringBuilder("One man, One vote");
//                                         01234567890123456
int subInd1a = banner2.indexOf("One");          // 0
int subInd1b = banner2.indexOf("One", 3);       // 9
int subInd2a = banner2.lastIndexOf("One");      // 9
int subInd2b = banner2.lastIndexOf("One", 10);  // 9
int subInd2c = banner2.lastIndexOf("One", 8);   // 0
int subInd2d = banner2.lastIndexOf("One", 2);   // 0
```

Extracting Substrings from String Builders

The `StringBuilder` class provides the following methods to extract substrings, which are also provided by the `String` class (p. 453).

Click here to view code image

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

`CharSequence subSequence(int start, int end)` From the `CharSequence` interface (p. 444).

Examples of extracting substrings:

Click here to view code image

```
StringBuilder bookName = new StringBuilder("Java Gems by Anonymous");
//                                          0123456789012345678901 2
String title  = bookName.substring(0,9);    // "Java Gems"
String author = bookName.substring(13);     // "Anonymous"
```

Constructing Strings from String Builders

The `StringBuilder` class overrides the `toString()` method from the `Object` class (see also the `CharSequence` interface, p. 444). It returns the contents of a string builder as a `String` object.

Click here to view code image

```
String toString()                    From the CharSequence interface (p. 444).
```

Click here to view code image

```
StringBuilder strBuilder = new StringBuilder("Build or not to build.");
String fromBuilder = strBuilder.toString();     // "Build or not to build."
```

Comparing String Builders

String builders implement the `Comparable<StringBuilder>` interface and can be compared lexicographically, analogous to `String` objects (p. 447).

Click here to view code image

```
int compareTo(StringBuilder anotherSB)
```

Here are some examples of string builder comparisons:

Click here to view code image

```
StringBuilder sb1 = new StringBuilder("abba");
StringBuilder sb2 = new StringBuilder("aha");

int compValue1 = sb1.compareTo(sb2);          // negative value => sb1 < sb2
int compValue2 = sb2.compareTo(sb1);          // positive value => sb2 > sb1
boolean isEqual  = sb1.compareTo(sb2) == 0;             // false
boolean flag     = sb1.toString().equals(sb2.toString()); // false
```

Modifying String Builders

Appending, inserting, replacing, and deleting characters automatically results in adjustment of the string builder's structure and capacity, if necessary. The indices passed as arguments in the methods must be equal to or greater than 0. An `IndexOutOfBoundsException` is thrown if an index is not valid.

Note that the methods in this subsection return the reference value of the modified string builder, making it convenient to chain calls to these methods.

Appending Characters to a String Builder

The overloaded method `append()` can be used to *append* characters at the *end* of a string builder.

Click here to view code image

```
StringBuilder append(Object obj)
```

The `obj` argument is converted to a string as if by the static method call `String.valueOf(obj)`, and this string is appended to the current string builder.

Click here to view code image

```
StringBuilder append(String str)
StringBuilder append(CharSequence charSeq)
StringBuilder append(CharSequence charSeq, int start, int end)
StringBuilder append(char[] charArray)
StringBuilder append(char[] charArray, int offset, int length)
StringBuilder append(char c)
```

Allow characters from various sources to be appended to the end of the current string builder.

Click here to view code image

```
StringBuilder append(boolean b)
StringBuilder append(int i)
StringBuilder append(long l)
StringBuilder append(float f)
StringBuilder append(double d)
```

Convert the primitive value of the argument to a string by applying the static method `String.valueOf()` to the argument, before appending the result to the string builder.

Inserting Characters in a String Builder

The overloaded method `insert()` can be used to *insert* characters at a *given offset* in a string builder.

Click here to view code image

```
StringBuilder insert(int offset, Object obj)
StringBuilder insert(int dstOffset, CharSequence seq)
StringBuilder insert(int dstOffset, CharSequence seq, int start, int end)
StringBuilder insert(int offset, String str)
StringBuilder insert(int offset, char[] charArray)
StringBuilder insert(int offset, type c)
```

In the methods above, *type* can be `char`, `boolean`, `int`, `long`, `float`, or `double`. The second argument is converted to a string, if necessary, by applying the static method `String.valueOf()`. The `offset` argument specifies where the characters are to be inserted in the string builder, and must be

> greater than or equal to 0. Note that the `offset` specifies the number of characters from the start of the string builder.

Replacing Characters in a String Builder

The following methods can be used to replace characters in a string builder:

> Click here to view code image
>
> ```
> void setCharAt(int index, char ch)
> ```
>
> Changes the character at a specified index in the string builder. An `IndexOutOfBoundsException` is thrown if the index is not valid. This method does not change the length of the string builder, and does *not* return a value.
>
> Click here to view code image
>
> ```
> StringBuilder replace(int start, int end, String replacement)
> ```
>
> Replaces the characters in a subsequence of the string builder with the characters in the specified `String`. The subsequence is defined by the `start` index (inclusive) and the `end` index (exclusive). It returns the modified string builder. If the `start` index is not valid (i.e., `start` index is negative, greater than `length()`, or greater than `end` index.), a `StringIndexOutOfBoundsException` is thrown.

Deleting Characters in a String Builder

The following methods can be used to delete characters from *specific positions* in a string builder:

> Click here to view code image
>
> ```
> StringBuilder deleteCharAt(int index)
> StringBuilder delete(int start, int end)
> ```
>
> The first method deletes a character at a specified index in the string builder, contracting the string builder by one character. The second method deletes a substring, which is specified by the `start` index (inclusive) and the `end` index (exclusive), contracting the string builder accordingly. If `start` index is equal to `end` index, no changes are made.

Reversing Characters in a String Builder

The following method in the class `StringBuilder` reverses the contents of a string builder:

```
StringBuilder reverse()
```

Examples of using methods that modify string builders:

Click here to view code image

```
StringBuilder builder = new StringBuilder("banana split");    // "banana split"
builder.delete(4,12);                                          // "bana"
builder.append(42);                                            // "bana42"
builder.insert(4,"na");                                        // "banana42"
builder.reverse();                                             // "24ananab"
builder.deleteCharAt(builder.length()-1);                      // "24anana"
builder.replace(0, 2, "b");                                    // "banana"
builder.append('s');                                           // "bananas"
```

All of the previously mentioned methods modify the contents of the string builder and return a reference value denoting the current string builder. This allows *chaining* of method calls. The method calls invoked on the string builder denoted by the reference `builder` can be chained as follows, giving the same result:

Click here to view code image

```
builder.delete(4,12).append(42).insert(4,"na").reverse()
       .deleteCharAt(builder.length()-1).replace(0, 2, "b")
       .append('s');                                            // "bananas"
```

The method calls in the chain are evaluated from left to right, so the previous chain of calls is interpreted as follows:

Click here to view code image

```
((((((builder.delete(4,12)).append(42)).insert(4,"na")).reverse())
      .deleteCharAt(builder.length()-1)).replace(0, 2, "b"))
      .append('s');                                             // "bananas"
```

Each method call returns the reference value of the modified string builder, which is then used to invoke the next method. The string builder remains denoted by the reference `builder`.

The compiler uses string builders to implement string concatenation with the `+` operator in `String`-valued non-constant expressions. The following code illustrates this optimization:

Click here to view code image

```
String theChosen = "U";
String str1 = 4 + theChosen + "Only";             // (1) Non-constant expression
```

The assignment statement at (1) is equivalent to the following code using a string builder:

Click here to view code image

```
String str2 = new StringBuilder().
                    append(4).append(theChosen).append("Only").toString(); // (2
```

The code at (2) does not create any temporary `String` objects when concatenating several strings, since a single `StringBuilder` object is modified and finally converted to a `String` object having the string content `"4UOnly"`.

Controlling String Builder Capacity

The following methods are exclusive to the `StringBuilder` class and can be used to control various capacity-related aspects of a string builder:

```
int capacity()
```

Returns the current capacity of the string builder, meaning the number of characters the current builder can accommodate without allocating a new, larger array to hold characters.

Click here to view code image

```
void ensureCapacity(int minCapacity)
```

Ensures that there is room for at least a `minCapacity` number of characters. It expands the string builder, depending on the current capacity of the builder.

```
void trimToSize()
```

Attempts to reduce the storage used for the character sequence. It may affect the capacity of the string builder.

Click here to view code image

```
void setLength(int newLength)
```

Ensures that the actual number of characters—that is, the length of the string builder—is exactly equal to the value of the `newLength` argument, which must be greater than or equal to 0. This operation can result in the string being truncated or padded with null characters (`'\u0000'`). This method affects the capacity of the string builder only if the value of the parameter `newLength` is greater than the current capacity.

Example 8.5 illustrates the various capacity-related methods of the

`StringBuilder` class. It is instructive to go through the output to see how these methods affect the length and the capacity of a string builder.

Example 8.5 *Controlling String Builder Capacity*

Click here to view code image

```
public class StringBuilderCapacity {
  public static void main(String[] args) {
    StringBuilder builder = new StringBuilder("No strings attached!");
    System.out.println("Builder contents: " + builder);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());
    System.out.println("Ensure capacity of 40");
    builder.ensureCapacity(40);
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Trim to size");
    builder.trimToSize();
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Set length to 10");
    builder.setLength(10);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder contents: " + builder);
    System.out.println("Set length to 0");
    builder.setLength(0);
    System.out.println("Builder is empty: " + (builder.length() == 0));
  }
}
```

Probable output from the program:

Click here to view code image

```
Builder contents: No strings attached!
Builder length:   20
Builder capacity: 36
Ensure capacity of 40
Builder capacity: 74
Trim to size
Builder length:   20
Builder capacity: 20
Set length to 10
Builder length:   10
Builder contents: No strings
Set length to 0
Builder is empty: true
```

Review Questions

8.5 Which expression will extract the substring `"kap"`, given the following declaration?

```
String str = "kakapo";
```

Select the one correct answer.

a. `str.substring(2, 2)`
b. `str.substring(2, 3)`
c. `str.substring(2, 4)`
d. `str.substring(2, 5)`
e. `str.substring(3, 3)`

8.6 What will be the result of attempting to compile and run the following code?

Click here to view code image

```
class MyClass {
  public static void main(String[] args) {
    String str1 = "str1";
    String str2 = "str2";
    String str3 = "str3";

    str1.concat(str2);
    System.out.println(str3.concat(str1));
  }
}
```

Select the one correct answer.

a. The code will fail to compile.
b. The program will print `str3str1str2` at runtime.
c. The program will print `str3` at runtime.
d. The program will print `str3str1` at runtime.
e. The program will print `str3str2` at runtime.

8.7 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class RefEq {
  public static void main(String[] args) {
    String s = "ab" + "12";
    String t = "ab" + 12;
    String u = new String("ab12");
    System.out.println((s==t) + " " + (s==u));
  }
}
```

Select the one correct answer.

a. The program will fail to compile.
b. The program will print `false false` at runtime.
c. The program will print `false true` at runtime.

d. The program will print `true false` at runtime.
e. The program will print `true true` at runtime.

8.8 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.
b. The program will print `false false`.
c. The program will print `true false`.
d. The program will print `true true`.
e. The program will fail to compile.
f. The program will compile, but it will throw an exception at runtime.

8.9 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class FunCharSeq {
  private static void putO(String s1) {
    s1 = s1.trim();
    s1 += "O";
  }

  public static void main(String[] args) {
    String s1 = " W ";
    putO(s1);
    s1.concat("W");
    System.out.println("|" + s1 + "|");
  }
}
```

Select the one correct answer.

a. `|WOW|`
b. `| W W|`
c. `|WO|`
d. `| W |`
e. The program will fail to compile.
f. The program will compile, but it will throw an exception at runtime.

8.10 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.
b. The program will print `false false`.
c. The program will print `true false`.
d. The program will print `true true`.
e. The program will fail to compile.
f. The program will compile, but it will throw an exception at runtime.

8.11 What will the following code print when run?

Click here to view code image

```
String s = "This is hard";
s = "-" + s.substring(s.indexOf(' '), s.indexOf(' ', s.indexOf(' ') + 1)
    .strip() + "-";
System.out.println(s);
```

Select the one correct answer.

a. `- is -`
b. `-is-`
c. `-is -`
d. `- is-`
e. `-s is h-`
f. `-sish-`

8.12 Given the following code:

Click here to view code image

```
String txt = """
  a
    b
  c
""";
int from = 0;
int to = txt.indexOf('\n');
String line = null;
while(to < txt.length()-1) {
  to = txt.indexOf('\n', from);
  line = txt.substring(from, to);
  System.out.print(line.length());
  from = to+1;
```

```
    }
```

The first line of the text block has two leading spaces, the second line has four leading spaces, and the third line has two leading spaces. There are no leading spaces on the line with the closing delimiter of the text block.

What is the result?

Select the one correct answer.

a. `353`
b. `3530`
c. `1510`
d. `131`

8.13 Which text block produces a single line of text with the following exact characters when printed?

```
"a""b"
```

Select the one correct answer.

a. `String txt = """"a""b\"""";`

b.
```
String txt = """"a""b"
""";
```

c.
```
String txt = """
"a""b"""";
```

d.
```
String txt = """
"a""b\"""";
```

e.
```
String txt = """
"a""b"
""";
```

f.
```
String txt = """
"a\""b"\""";
```

8.14 Which of the following statements are true about text blocks? Select the two correct answers.

a. Content of a text block starts immediately after the line terminator of the opening delimiter.
b. Content of a text block ends immediately after the line terminator of the closing delimiter.
c. A text block is a subtype of the `String` class.
d. All leading whitespace is removed from each line in the text block.
e. All trailing whitespace is removed from each line in the text block.
f. Incidental whitespace is not removed from each line in the

text block.

8.15 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    String str = "hello";
    StringBuilder sb = new StringBuilder(str);
    sb.reverse();
    if (str == sb) System.out.println("a");
    if (str.equals(sb)) System.out.println("b");
    if (sb.equals(str)) System.out.println("c");
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw an exception at runtime.

c. The program will compile, but it will not print anything.

d. The program will compile, and will print `abc`.

e. The program will compile, and will print `bc`.

f. The program will compile, and will print `a`.

g. The program will compile, and will print `b`.

h. The program will compile, and will print `c`.

8.16 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("Have a nice day");
    sb.setLength(6);
    System.out.println(sb);
  }
}
```

Select the one correct answer.

a. The code will fail to compile because there is no method named `setLength` in the `StringBuilder` class.

b. The code will fail to compile because the `StringBuilder` reference `sb` is not a legal argument to the `println()` method.

c. The program will throw a `StringIndexOutOfBoundsException` at runtime.

d. The program will print `Have a nice day` at runtime.

e. The program will print `Have a` at runtime.
f. The program will print `ce day` at runtime.

8.17 Which statement is true about the following code, where the argument string `"  1234  "` has two leading and two trailing spaces?

Click here to view code image

```
public class RQ_8_24 {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("  1234  ");
    sb.trimToSize();
    sb.append("!");
    sb.reverse();
    sb.setLength(5);
    System.out.println("|" + sb + "|");
  }
}
```

Select the one correct answer.

a. The program will print `|4321!|`.
b. The program will print `|!1234|`.
c. The program will print `|!  43|`.
d. The program will print `|1234!|`.
e. The program will print `|!4321|`.
f. The program will fail to compile.

8.18 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class PeskyCharSeq {
  public static void main (String[] args)  {
    StringBuilder sb1  = new StringBuilder("WOW");
    StringBuilder sb2  = new StringBuilder(sb1);
    System.out.println((sb1==sb2) + " " + sb1.equals(sb2));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.
b. The program will print `false false`.
c. The program will print `true false`.
d. The program will print `true true`.
e. The program will fail to compile.
f. The program will compile, but it will throw an exception at runtime.

8.19 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class MoreCharSeq {
  public static void main (String[] args) {
    String s1 = "WOW";
    StringBuilder s2 = new StringBuilder(s1);
    String s3 = new String(s2);
    System.out.println((s1.hashCode() == s2.hashCode()) + " " +
                       (s1.hashCode() == s3.hashCode()));
  }
}
```

Select the one correct answer.

- **a.** The program will print `false true`.
- **b.** The program will print `false false`.
- **c.** The program will print `true false`.
- **d.** The program will print `true true`.
- **e.** The program will fail to compile.
- **f.** The program will compile, but it will throw an exception at runtime.

8.20 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Appendage {
  private static void putO(StringBuilder s1) {
    s1.append("O");
  }

  public static void main(String[] args) {
    StringBuilder s1 = new StringBuilder("W");
    putO(s1);
    s1.append("W!");
    System.out.println(s1);
  }
}
```

Select the one correct answer.

- **a.** The program will print `WW!`.
- **b.** The program will print `WOW!`.
- **c.** The program will print `W`.
- **d.** The program will print `WO`.
- **e.** The program will fail to compile.
- **f.** The program will compile, but it will throw an exception at runtime.

8.21 What will the following code print when run?

Click here to view code image

```
StringBuilder text = new StringBuilder();
text.append("12");
text.insert(1, "34");
text.delete(1, 1);
text.replace(0, 1, "");
System.out.println(text);
```

Select the one correct answer.

a. `2`
b. `12`
c. `32`
d. `34`
e. `42`
f. `134`
g. `234`
h. `312`
i. `342`
j. `412`

8.22 What will the following code print when run?

Click here to view code image

```
StringBuilder text = new StringBuilder();
text.append("42");
text.delete(1,2);
System.out.println(text.toString() + (text.capacity() + text.length()));
```

Select the one correct answer.

a. `416`
b. `417`
c. `4161`
d. `4171`
e. `410`
f. `411`
g. `4101`
h. `4111`

8.6 The `Math` Class

The `final` class `java.lang.Math` defines a set of `static` methods to support common mathematical functions, including functions for rounding numbers, finding the maximum and minimum of two numbers, calculating logarithms and exponentiation, performing exact arithmetic, generating pseudorandom numbers, and much more. The `Math` class is a utility class and cannot be instantiated.

The `final` class `Math` provides constants to represent the value of *e*, the base of the natural logarithms, and the value π (*pi*), the ratio of the circumference of a circle to its diameter:

```
Math.E
Math.PI
```

Miscellaneous Rounding Functions

> Click here to view code image
>
> ```
> static int    abs(int i)
> static long   abs(long l)
> static float  abs(float f)
> static double abs(double d)
> ```
>
> The overloaded method `abs()` returns the absolute value of the argument. For a non-negative argument, the argument is returned. For a negative argument, the negation of the argument is returned.
>
> Click here to view code image
>
> ```
> static int    min(int a, int b)
> static long   min(long a, long b)
> static float  min(float a, float b)
> static double min(double a, double b)
> ```
>
> The overloaded method `min()` returns the smaller of the two values `a` and `b` for any numeric type.
>
> Click here to view code image
>
> ```
> static int    max(int a, int b)
> static long   max(long a, long b)
> static float  max(float a, float b)
> static double max(double a, double b)
> ```
>
> The overloaded method `max()` returns the greater of the two values `a` and `b` for any numeric type.

The following code illustrates the use of these methods from the `Math` class:

Click here to view code image

```
long   ll = Math.abs(2022L);              // 2022L
double dd = Math.abs(-Math.PI);           // 3.141592653589793

double d1 = Math.min(Math.PI, Math.E);    // 2.718281828459045
long   m1 = Math.max(1984L, 2022L);       // 2022L
int    i1 = (int) Math.max(3.0, 4);       // Cast required.
```

Note the cast required in the last example. The method with the signature `max(double, double)` is executed, with implicit conversion of the `int` argument to a `double`. Since this method returns a `double`, it must be explicitly cast to an `int` in order to assign it to an `int` variable.

> Click here to view code image
>
> ```
> static double ceil(double d)
> ```
>
> Returns the *smallest* (*closest to negative infinity*) `double` value that is *greater than or equal to* the argument `d`, and is equal to a mathematical integer.
>
> Click here to view code image
>
> ```
> static double floor(double d)
> ```
>
> Returns the *largest* (*closest to positive infinity*) `double` value that is *less than or equal to* the argument `d`, and is equal to a mathematical integer. Note that `Math.ceil(d)` is exactly the value of `-Math.floor(-d)`.
>
> Click here to view code image
>
> ```
> static int round(float f)
> static long round(double d)
> ```
>
> The overloaded method `round()` returns the integer closest to the argument. This is equivalent to adding 0.5 to the argument, taking the floor of the result, and casting it to the return type. This is not the same as rounding to a specific number of decimal places, as the name of the method might suggest.
>
> If the fractional part of a *positive* argument is *less than* 0.5, then the result returned is the same as `Math.floor()`. If the fractional part of a positive argument is *greater than or equal to* 0.5, then the result returned is the same as `Math.ceil()`.
>
> If the fractional part of a *negative* argument is *less than or equal to* 0.5, then the result returned is the same as `Math.ceil()`. If the fractional part of a negative argument is *greater than* 0.5, then the result returned is the same as `Math.floor()`.

It is important to note the result obtained on negative arguments, keeping in mind that a negative number whose absolute value is less than that of another negative number is actually greater than the other number (e.g., –3.2 is greater than –4.7). Compare also the results returned by these methods, shown in Table 8.3, p. 480.

Click here to view code image

```
double upPI    = Math.ceil(3.14);              // 4.0
double downPI  = Math.floor(3.14);             // 3.0
long   roundPI = Math.round(3.14);             // 3L

double upNegPI    = Math.ceil(-3.14);          // -3.0
double downNegPI  = Math.floor(-3.14);         // -4.0
long   roundNegPI = Math.round(-3.14);         // -3L
```

Table 8.3 *Applying Rounding Functions*

Arg [illegible]											
ceil: [illegible]											
floor: [illegible]											
round: [illegible]											
Arg [illegible]											
ceil: [illegible]											
floor: [illegible]											
round: [illegible]											

Exponential Functions

> Click here to view code image
>
> `static double pow(double d1, double d2)`
>
> Returns the value of `d1` raised to the power of `d2` (i.e., $d1^{d2}$).
>
> Click here to view code image
>
> `static double exp(double d)`
>
> Returns the exponential number *e* raised to the power of `d` (i.e., e^{d}).
>
> Click here to view code image
>
> `static double log(double d)`
>
> Returns the natural logarithm (base *e*) of `d` (i.e., $log_{e}d$).
>
> Click here to view code image
>
> `static double sqrt(double d)`
>
> Returns the square root of `d` (i.e., $d^{0.5}$). For a `NaN` or a negative argument, the result is a `NaN`.

Some examples of exponential functions:

Click here to view code image

```
double r = Math.pow(2.0, 4.0);            // 16.0
double v = Math.exp(2.0);                 // 7.38905609893065
double l = Math.log(Math.E);              // 0.9999999999999981
double c = Math.sqrt(25.0);               // 5.0
```

Exact Integer Arithmetic Functions

Integer arithmetic operators (like `+`, `-`, `*`) do not report *overflow errors*—that is, the integer values *wrap around* (§2.8, p. 59). However, if it is important to detect overflow errors, the `Math` class provides methods that report overflow errors by throwing an `ArithmeticException` when performing integer arithmetic operations. The relevant methods have the postfix `"Exact"` in their name, and are shown below.

The code snippets below illustrate exact integer arithmetic operations provided by the `Math` class, where we assume the field `System.out` is statically imported. The results computed at (1a), (2a), and (3a) are incorrect due to overflow, but the standard arithmetic operators do not report overflow errors. (1b), (2b), and (3b) use exact arithmetic operations and throw an `ArithmeticException` when an overflow is detected. (1c), (2c), and (3c) compute correct results as there is no overflow in performing the operation.

Click here to view code image

```
out.println(Integer.MAX_VALUE + 1);                 // (1a) -2147483648
out.println(Math.addExact(Integer.MAX_VALUE, 1));   // (1b) ArithmeticExceptio
out.println(Math.addExact(1_000_000, 1_000));       // (1c) 1001000
out.println(Integer.MAX_VALUE * 100);               // (2a) -100
out.println(
     Math.multiplyExact(Integer.MAX_VALUE, 100)     // (2b) ArithmeticExceptio
);
out.println(Math.multiplyExact(1_000_000, 1_000));  // (2c) 1000000000

out.println((int)Long.MAX_VALUE);                   // (3a) -1
out.println(Math.toIntExact(Long.MAX_VALUE));       // (3b) ArithmeticExceptio
out.println(Math.toIntExact(1_000_000));            // (3c) 1000000
```

Selected exact integer arithmetic methods in the `Math` class are shown below.

> Click here to view code image
>
> ```
> static int absExact(int a)
> static long absExact(long a)
> ```
>
> Return the mathematical absolute value of an `int` or a `long` value if it is exactly representable as an `int` or a `long`, throwing `ArithmeticException` if the argument is `Integer.MIN_VALUE` or `Long.MIN_VALUE`, as these argument values would overflow the positive `int` or `long` range, respectively.
>
> Click here to view code image

```
static int  addExact(int x, int y)
static long addExact(long x, long y)
static int  subtractExact(int x, int y)
static long subtractExact(long x, long y)
```

Return the sum or difference of the given arguments, throwing `Arithmetic-Exception` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int  multiplyExact(int x, int y)
static long multiplyExact(long x, int y)
static long multiplyExact(long x, long y)
```

Return the product of the arguments, throwing `ArithmeticException` if the result overflows an `int` in the first method or a `long` in the last two methods.

Click here to view code image

```
static int  incrementExact(int a)
static long incrementExact(long a)
static int  decrementExact(int a)
static long decrementExact(long a)
```

Return the argument incremented or decremented by 1, throwing `Arithmetic-Exception` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int  negateExact(int a)
static long negateExact(long a)
```

Return the negation of the argument, throwing `ArithmeticException` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int toIntExact(long value)
```

Returns the value of the `long` argument as an `int`, throwing `ArithmeticException` if the value overflows an `int`.

Pseudorandom Number Generator

The class `Math` provides the `random()` method for generating pseudorandom numbers of type `double` that are in the open interval `[0.0, 1.0)`. The `Random` class (p. 482) should also be considered, as it is more versatile and easier to use.

```
static double random()
```

> Returns a random number greater than or equal to 0.0 and less than 1.0, where the value is selected randomly from the range according to a uniform distribution.

We can simulate a dice roll as follows:

Click here to view code image

```
int diceValue = 1 + (int)(Math.random() * 6.0);  // A dice roll in range [1 ..
```

The dice value will always be in the interval `[1, 6]`, depending on the pseudorandom number returned by the `Math.random()` method, as we can see from the sample dice rolls below.

Click here to view code image

```
Math.random():     {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}
Multiply by 6:     {0.0, 0.6, 1.2, 1.8, 2.4, 3.0, 3,6, 4.2, 4.8, 5.4}
Convert to int:    {0,   0,   1,   1,   2,   3,   3,   4,   4,   5}
Add offset 1:      {1,   1,   2,   2,   3,   4,   4,   5,   5,   6}
```

8.7 The `Random` Class

In computer games and simulations we often need to generate a sequence of random numbers. For a dice game it is necessary to simulate rolling the dice—that is, generating a dice roll value between 1 and 6. Ideally, the probability of a given value is the same (one-sixth) for each value on a die. The numbers generated this way are called *random numbers*. To generate random numbers with the help of a program is not possible, as a program can only compute values, not pick them randomly. To guarantee equal probability for all cases is very difficult, so we have to settle for *pseudorandom numbers*—that is, a sequence of numbers that closely approximates a sequence of random numbers. A lot of research has gone into finding mathematical formulae that compute good approximations to random numbers.

The `java.util.Random` class implements the `java.util.random.RandomGenerator` interface that defines the common protocol for pseudorandom number generators (PRNGs) in Java.

Generating *numeric streams of pseudorandom values* using the `Random` class is covered with the discussion on creating streams (§16.4, p. 900).

The following constructors can be used to create a new random number generator, optionally specifying a *seed* value (p. 484).

```
Random()
Random(long seed)
```

The `Random` class implements PRNGs for different primitive data types. The appropriate *next* method can be called on a `Random` object to obtain the next pseudorandom value.

> Click here to view code image
>
> ```
> int     nextInt()
> int     nextInt(int bound)
> long    nextLong()
> float   nextFloat()
> double  nextDouble()
> boolean nextBoolean()
> ```
>
> Return the next pseudorandom, uniformly distributed value that is either `int`, `int` between `0` and `bound` (exclusive), `long`, `float` between `0.0f` and `1.0f` (exclusive), `double` between `0.0d` and `1.0d` (exclusive), or `true/false`, respectively.

Here we will concentrate on a pseudorandom generator for `int` values, but first we need to create an object of the `Random` class:

Click here to view code image

```
Random generator = new Random();
```

We can then call the method `nextInt()` repeatedly on this object every time we need a new random `int` value:

Click here to view code image

```
int number = generator.nextInt();
```

Each call to the method will return a random integer in the interval $[-2^{31}, 2^{31} - 1]$, which is the range of the `int` data type.

Determining the Range

We are often interested in generating random numbers in a particular range. For example, the following code will return a random number in the interval `[0, 10]`:

Click here to view code image

```
number = generator.nextInt(11);      // Random integer in the interval [0, 10]
```

If the bound value is `n`, the integer returned is in the interval `[0, n-1]`. By supplying a new bound value to the `nextInt()` method, we can change the upper bound of the original interval.

If we want to shift the interval, we can add (or subtract) an offset from the value returned by the `nextInt()` method. In the code below, values generated in the original interval `[0, 10]` will now lie in the interval `[2, 12]`—that is, the offset `2` maps the interval `[0, 10]` onto the interval `[2, 12]`:

Click here to view code image

```
number = 2 + generator.nextInt(11);  // Random integer in the interval [2, 12]
```

If we want the values in the interval to have a distance greater than 1, we can multiply the value generated by a distance value:

Click here to view code image

```
number = 2 + 3*generator.nextInt(5); // Random integer in the set {2, 5, 8, 11
```

With a distance value of `3`, the expression `3*generator.nextInt(5)` always returns a value in the set `{0, 3, 6, 9, 12}`, and an offset of `2` ensures that the variable `number` is assigned a value from the set `{2, 5, 8, 11, 14}`.

We can simulate a dice roll as follows:

Click here to view code image

```
int diceValue = 1 + generator.nextInt(6);  // A dice value in the interval [1,
```

The expression `generator.nextInt(6)` always returns a value in the set `{0, 1, 2, 3, 4, 5}`, and an offset of `1` ensures that the variable `diceValue` is assigned a value from the set `{1, 2, 3, 4, 5, 6}`.

Generating the Same Sequence of Pseudorandom Numbers

The way in which we have used the pseudorandom number generator up to now cannot guarantee the same sequence of pseudorandom numbers each time the program is run. This is because the pseudorandom number generator is based on the time of the system clock. This is obviously different each time the program is run. If we want to generate the same sequence of pseudorandom numbers each time the program is run, we can specify a *seed* in the call to the `Random` constructor:

Click here to view code image

```
Random persistentGenerator = new Random(31);
```

In the declaration above, the seed is the prime number 31. The seed is usually a prime number (i.e., a number that is only divisible by itself or one), as such numbers are highly suitable for implementing good, viable pseudorandom number generators.

8.8 Using Big Numbers

There is a need for greater precision in arithmetic calculations than the precision offered by arithmetic operators performing calculations on finite-precision values of numeric primitive types. This is especially true for calculations involving financial or monetary values, where accumulation of rounding errors and precision is paramount. The classes `BigInteger` and `BigDecimal` in the `java.math` package support arbitrary-precision integers and arbitrary-precision decimal numbers, respectively, and provide methods for arbitrary-precision computations (Table 8.4). Both classes extend the `java.lang.Number` abstract class (Figure 8.1).

Table 8.4 *Big Number Classes*

Big number classes in the `jav`	**Description**
`BigDecimal extends Number`	A class that represents immutable, arbitrary-precision, signed decimal numbers
`BigInteger extends Number`	A class that represents immutable, arbitrary-precision signed integers

A `BigDecimal` is represented as an arbitrary-precision *unscaled integer value* and a 32-bit *integer scale*. For a non-negative unscaled value, the scale represents the number of digits to the right of the decimal point. For a negative unscaled value, the number represented is (`unscaled value * 10`$^{\text{-scale}}$). For example:

Click here to view code image

```
The BigDecimal  3.14   has unscaled value =  314   and scale = 2.
The BigDecimal -3.1415 has unscaled value = -31415 and scale = 4.
```

The `BigDecimal` class can represent very large and very small decimal numbers with very high precision. The class provides methods that allow control over scale and rounding behavior for arithmetic operations. In addition, the class provides support for converting to different representations and comparing `BigDecimal` values.

Formatting, parsing, and rounding of `BigDecimal` values is covered with formatting of numbers and currency (§18.5, p. 1116).

A `BigInteger` represents an arbitrary-precision signed integer value in base 10 and two's-complement notation, which is internally implemented by an arbitrary-size `int` array. The `BigInteger` class has methods analagous to the `BigDecimal` class for arithmetic calculations, conversions, and comparison.

Here we will only provide a few examples of using the `BigDecimal` class. The API for the `BigDecimal` and `BigInteger` classes in the `java.math` package should be consulted, both for details on features presented here and additional features provided for high-precision manipulation of big numbers.

`BigDecimal` Constants

The class `BigDecimal` provides the following ready-made constants corresponding to the decimal values 0, 1, and 10, that are represented with a scale of 0:

```
BigDecimal.ZERO
BigDecimal.ONE
BigDecimal.TEN
```

Constructing `BigDecimal` Numbers

Selected constructors of the `BigDecimal` class are shown below.

```
BigDecimal(int value)
BigDecimal(long value)
BigDecimal(double value)
```

Create a `BigDecimal` with the decimal number representation of the specified value. Note that these constructors can result in a loss of precision if the specified numerical value does not have an exact representation.

Click here to view code image

```
BigDecimal(String strValue)
```

Creates a `BigDecimal` from the string representation of a numerical value. Recommended as the preferred way of creating `BigDecimal` numbers.

The following `valueOf()` methods also create `BigDecimal` numbers:

> Click here to view code image
>
> ```
> static BigDecimal valueOf(long value)
> static BigDecimal valueOf(double value)
> ```
>
> Create a `BigDecimal` from the specified numerical `value`. The second method uses the string representation of the `double` `value` obtained from the `Double.toString(double)` method to create a `BigDecimal`.

The following code creates `BigDecimal` values and also shows the string representation of the `BigDecimal` numbers created. It shows that the constructor `BigDecimal(String)` or the static `valueOf()` methods are preferable when creating `BigDecimal` values, especially for decimal values that cannot be represented as exact `double` values.

Click here to view code image

```
BigDecimal dTobd   = new BigDecimal(0.7);         // 0.6999999999999999555910..
BigDecimal strTobd = new BigDecimal("0.7");       // 0.7
BigDecimal valTobd = BigDecimal.valueOf(0.7);     // 0.7
```

Computing with `BigDecimal` Numbers

Selected methods that perform common arithmetic operations with `BigDecimal` numbers are given below.

The limitations of arithmetic operators on decimal values are shown by the code below. An incorrect result may be computed if the representation of a decimal value is inexact in the format of the `double` primitive type.

Click here to view code image

```
double d1 = 0.70;
double d2 = 0.10;
System.out.println(d1 + d2);
```

Output: `0.7999999999999999`

However, using `BigDecimal` values gives the correct result:

Click here to view code image

```
BigDecimal bd1b = new BigDecimal("0.70");
BigDecimal bd2b = new BigDecimal("0.10");
System.out.println(bd1b.add(bd2b));
```

Output: `0.80`

Finally, here is a simple arithmetic calculation using `BigDecimal` numbers to compute the total cost of 10^3 items, priced at $2.99 each and 25% sales tax.

Click here to view code image

```
BigDecimal price     = new BigDecimal("2.99");
BigDecimal tax       = new BigDecimal("0.25");
BigDecimal quantity  = BigDecimal.TEN.pow(3);
BigDecimal totalCost = price.add(price.multiply(tax)).multiply(quantity);
System.out.println(totalCost);
```

Output: `3737.5000`

It is worth keeping in mind that `BigDecimal` numbers are immutable, and not to fall into the pitfall shown in the code below:

Click here to view code image

```
BigDecimal sum = BigDecimal.ZERO;
sum.add(BigDecimal.ONE);            // sum is not updated! Returns new BigDecim
```

Selected methods for computing with `BigDecimal` numbers:

Click here to view code image

```
BigDecimal add(BigDecimal val)
BigDecimal subtract(BigDecimal val)
BigDecimal multiply(BigDecimal val)
BigDecimal divide(BigDecimal val)
BigDecimal remainder(BigDecimal val)
```

Return a `BigDecimal` whose value represents the result of performing the operation `(this + val)`, `(this - val)`, `(this * val)`, `(this / val)`, or `(this % val)`, respectively.

```
BigDecimal abs()
```

Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`.

```
BigDecimal negate()
```

Returns a `BigDecimal` whose value is `(-this)`.

```
BigDecimal pow(int n)
```

Returns a `BigDecimal` whose value is (this^n).

Review Questions

8.23 Given the following program, which lines will print `11` exactly?

Click here to view code image

```
class MyClass {
    public static void main(String[] args) {
        double v = 10.5;

        System.out.println(Math.ceil(v));        // (1)
        System.out.println(Math.round(v));       // (2)
        System.out.println(Math.ceil(v));  // (1)
        System.out.println(Math.round(v));  // (2)
        System.out.println(Math.floor(v));       // (3)
        System.out.println((int) Math.ceil(v));  // (4)
        System.out.println((int) Math.floor(v)); // (5)
    }
}
```

Select the two correct answers.

- **a.** The line at (1).
- **b.** The line at (2).
- **c.** The line at (3).
- **d.** The line at (4).
- **e.** The line at (5).

8.24 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Round {
    public static void main(String[] args) {
        System.out.println(Math.round(-0.5) + " " + Math.round(0.5));
    }
};
```

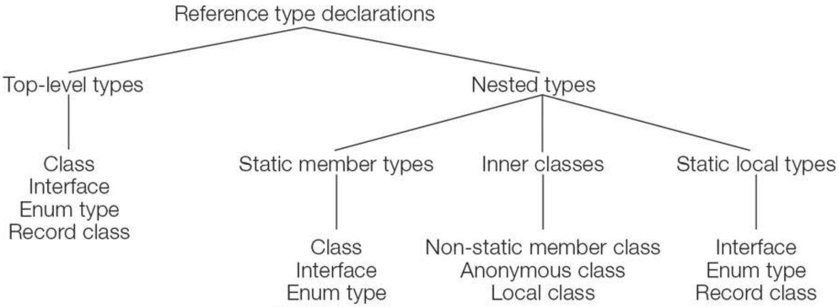
Select the one correct answer.

- **a.** `0 0`
- **b.** `0 1`
- **c.** `-1 0`
- **d.** `-1 1`
- **e.** None of the above

8.25 Which of the following statements are true about the expression `((int)(Math.random()*4))`?

Select the three correct answers.

a. It may evaluate to a negative number.
b. It may evaluate to the number 0.
c. The probability of it evaluating to the number 1 or the number 2 is the same.
d. It may evaluate to the number 3.
e. It may evaluate to the number 4.


Reference type declarations
Top-level types
Nested types
Class
Interface
Enum type
Record class
Static member types
Inner classes
Static local types
Class
Interface
Enum type
Record class
Non-static member class
Anonymous class
Local class
Interface
Enum type
Record class

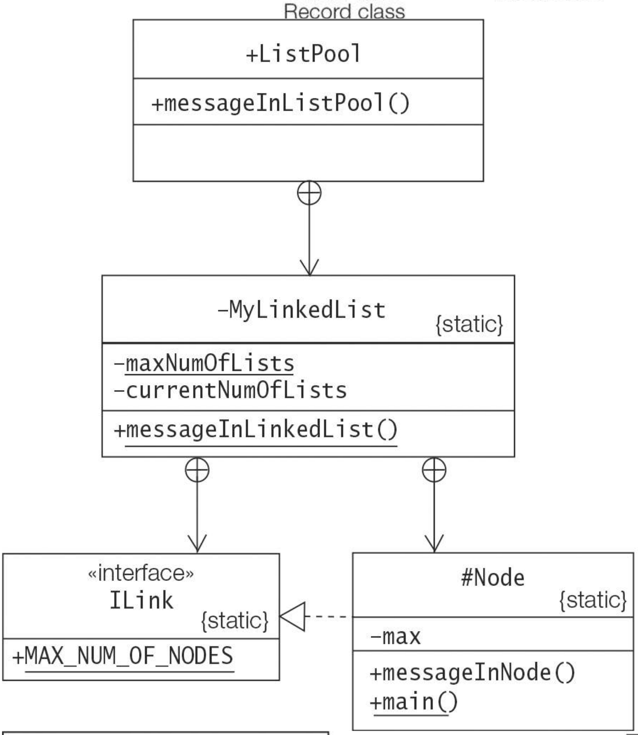
Accessibility:

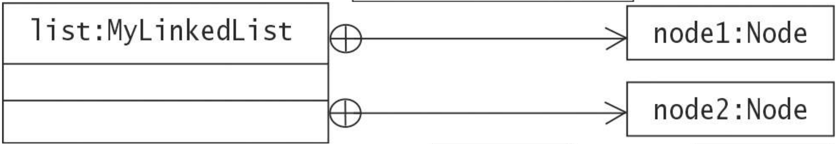
`+ : public`

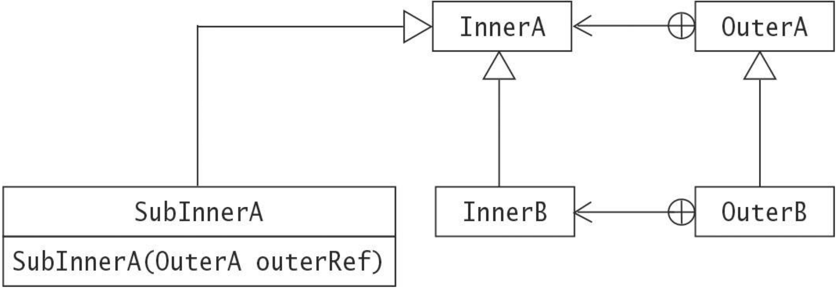
`- : private`

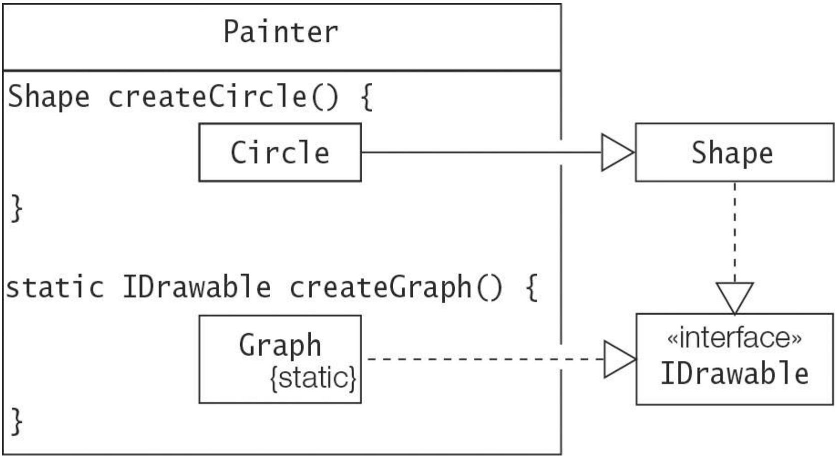
`# : protected`

None `: package`


+ListPool
+messageInListPool()
-MyLinkedList
{static}
-maxNumOfLists
-currentNumOfLists
+messageInLinkedList()
«interface»
ILink
{static}
+MAX_NUM_OF_NODES
#Node
{static}
-max
+messageInNode()
+main()


list:MyLinkedList
node1:Node
node2:Node


InnerA
OuterA
SubInnerA
SubInnerA(OuterA outerRef)
InnerB
OuterB


Painter
Shape createCircle() {
Circle
}
static IDrawable createGraph() {
Graph
{static}
}
Shape
«interface»
IDrawable

Nested Type Declarations 9

Chapter Topics

- Categorizing nested classes:
    - ❍ Static member type which can be a class, an enum type, a record class, or an interface
    - ❍ Inner class which can be a non-static member class, a local class, or an anonymous class
    - ❍ Static local type which can be an interface, an enum type, or a record class
- Understanding the salient aspects of nested types:
    - ❍ The context in which they can be defined
    - ❍ Which accessibility modifiers are valid for nested

types

    - Whether an instance of the enclosing context is associated with an instance of the nested class
    - Which entities a nested type can access in its enclosing context and in its inheritance hierarchy
    - Whether both static and non-static members can be defined in a nested type

- Importing and using nested types
- Instantiating non-static member classes using the *enclosing_object_reference*`.new` syntax
- Accessing members in the enclosing context of inner classes using the *enclosing_class_name*`.this` syntax
- Implementing anonymous classes by extending an existing class or by implementing an interface

Java SE 17 Developer Exam O	
§9.1, p. 491 to §9.6, p. 521	[9.1] Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection ○ *Only nested types are covered in this chapter.* ○ *For object lifecycle and garbage collection, see Chapter 10, p. 531.*

Java SE 11 Developer Exam O	
§9.1, p. 491 to §9.6, p. 521	[9.1] Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection) ○ *Only nested types are covered in this chapter.* ○ *For object lifecycle and garbage collection, see Chapter 10, p. 531.*

This chapter covers the different kinds of nested type declarations—that is, type declarations that can be declared inside a language construct such as a class, an interface, or even a method. In particular, we look at declaring, instantiating, and using such types. Since they are nested, we consider the rules for accessing entities in their enclosing context and in their inheritance hierarchy. Please hold on, as we first introduce the terminology for these nested types.

9.1 Overview of Nested Type Declarations

A *type declaration* allows a *new reference type* to be defined. A type declaration can either be a *top-level type declaration* or a *nested type declaration*. Figure 9.1 gives an overview of the different kinds of type declarations that can be defined in Java.

Figure 9.1 *Overview of Type Declarations*

A *top-level type declaration* is a type declaration that is *not* defined inside another type declaration. A top-level type declaration can be any one of the following: a *top-level class*, a *top-level interface*, a *top-level enum type*, or a *top-level record class*. We invariably use the shorter names *class*, *interface*, *enum type*, and *record class* when it is clear we are not referring to their nested counterparts.

A *nested type declaration* is a type declaration that is defined inside another declaration. There are three categories of nested types:

- Static member types (a.k.a. *static nested types*)
- Inner classes (a.k.a. *non-static nested classes*)
- Static local types (a.k.a. *static local nested types*)

As the name implies, *static member types* can be declared as a `static` *member* of either a top-level type or a nested type declaration. There are four kinds of static member types:

- Static member class (a.k.a. *static nested class*)
- Static member interface (a.k.a. *nested interface*)
- Static member enum type (a.k.a. *nested enum type*)
- Static member record class (a.k.a. *nested record class*)

A `static` member class or record class can be instantiated like any ordinary top-level class or record class, using its *qualified name* when calling the constructor with the `new` operator. It is not possible to instantiate an enum type or an interface.

Inner classes are non-`static` nested classes. There are three kinds of inner classes:

- Non-`static` member class
- Local class
- Anonymous class

Inner classes differ from `static` member classes in one important aspect: An instance of an inner class has an instance of the enclosing class (called the *immediately enclosing instance*) associated with it when the inner class is declared in a non-static context. An instance of an inner class can access the members of its enclosing instance by their simple names.

Non-static member classes are inner classes that are defined as instance members of other type declarations, just as fields and instance methods are defined in a class.

Local (normal) classes are non-`static` inner classes that can be defined in a block—a local block, a method body, an initializer block—both in static and non-static context, just as local variables can be defined in a block.

Static local types are defined in a block, but are *implicitly* `static` as opposed to local classes that are non-`static`. A static local type can be defined both in static and non-static context, just as local classes can be defined in a block. There are three kinds of static local types:

- Static local interface
- Static local enum type
- Static local record class

Anonymous classes are inner classes that can be defined as expressions and in expression statements, both in static and non-static context, and instantiated *on the fly*.

In Figure 9.1 we see that there are four kinds of *nested classes* (static member classes, non-`static` member classes, local classes, anonymous classes), two kinds of *nested interfaces* (static member interfaces, static local interfaces), two kinds of *nested enum types* (static member enum types, static local enum types), and two kinds of *nested record classes* (static member record classes, static local record classes)—all defined by the context in which these nested types are declared.

Given the terminology introduced for nested types in Figure 9.1, a *member type declaration* can be any one of the following nested types: a `static` member type (class, interface, enum type, record class) or a non-`static` member class. Note that local classes, anonymous classes, local interfaces, local enum types, and local record classes are *not* member type declarations, as they *cannot* be declared as a *member* of a type declaration.

Example 9.1 *Overview of Type Declarations*

Click here to view code image

```
class TLC {                       // (1) Top-level class

  // Static member types:
  static class     SMC {}         // (2) Static member class
         interface SMI {}         // (3) Static member interface
         enum      SME {}         // (4) Static member enum
         record    SMR() {}       // (5) Static member record

  // Non-static member class:
  class NSMC {}                   // (6) Inner class

  // Local types in non-static context (analogous for static context):
  void nsm() {                    // Non-static method
    class     LC {}               // (7) Local class (inner class)
    interface SLI {}              // (8) Static local interface
    enum      SLE {}              // (9) Static local enum
    record    SLR() {}            // (10) Static local record
  }

  // Anonymous classes (here defined as initializer expressions):
         SMC nsf = new SMC() {};  // (11) Inner class in non-static context
  static SMI  sf = new SMI() {};  // (12) Inner class in static context
}
```

Skeletal code for nested types is shown in Example 9.1. Table 9.1 presents a summary of various aspects relating to nested types. Subsequent sections on each nested type elaborate on the summary presented in this table. (*N/A* in the table means “*not applicable*”.)

- The *Type* column lists the different kinds of types that can be declared.
- The *Declaration context* column lists the lexical context in which a type can be declared.
- The *Access modifiers* column indicates what access can be specified for the type.
- The *Enclosing instance* column specifies whether an enclosing instance is associated with an instance of the type.
- The *Direct access to enclosing context* column lists what is directly accessible in the enclosing context from within the type.

Generic nested classes and interfaces are discussed in §11.13, p. 633. It is not possible to declare a generic enum type (§11.13, p. 635).

Locks on nested classes are discussed in §22.4, p. 1391.

Nested types can be regarded as a form of encapsulation, enforcing relationships between types by greater proximity. They allow structuring of types and a special binding relationship between a nested object and its enclosing instance. Used judiciously, they can be beneficial, but unrestrained use of nested types can easily result in unreadable code.

A word about the examples in this chapter: They are concocted to illustrate various aspects of nested types, and are not solutions to any well-defined or meaningful problems.

Table 9.1 *Various Aspects of Type Declarations*

[illegible]closing context
[illegible], enum type, or record class *(Top-level types)*
[illegible] type (*Static* [illegible]) are implicitly `public`
[illegible] type or a nested type
[illegible] context plus `final` or effectively `final` local variables
[illegible] context plus `final` or effectively `final` local variables
[illegible] plus `final` or effectively `final` local variables
[illegible] context plus `final` or effectively `final` local variables
[illegible] (*Static local types*)

9.2 Static Member Types

Declaring Static Member Types

Static member types can be declared in top-level type declarations, or within other nested types. For all intents and purposes, a `static` member type is very much like a top-level type.

A static member class, enum type, record class, or *interface* has the same declarations as those allowed in a top-level class, enum type, record class, or interface type, respectively. A `static` member class is declared with the keyword `static`, except when declared in an interface where a `static` member class is considered implicitly `static`. Static member enum types, record classes and interfaces are considered implicitly `static`, and the keyword `static` can be omitted.

Any access level (`public`, `protected`, package, `private`) can be specified for a `static` member type, except when declared in interfaces, where `public` access is implied for member type declarations.

Although the discussion in this section is primarily about static member classes and interfaces, it is also applicable to static member enum types and record classes.

In Example 9.2, the top-level class `ListPool` at (1) declares the `static` member class `MyLinkedList` at (2), which in turn defines a `static` member interface `ILink` at (3) and a `static` member class `BiNode` at (4). The `static` member class `BiNode` at (4) implements the `static` member interface `IBiLink` declared at (7). Note that each `static` member class is defined as `static`, just like `static` variables and methods in a top-level class.

In Example 9.2, an attempt to declare the `static` member class `Traversal` with `private` access at (8) in the interface `IBiLink` results in a compile-time error, as only `public` access is permitted for interface members. Since the class `BiTraversal` at (9) is defined in an interface; it is implicitly `public` and `static`, and *not* a non-`static` member class with package access.

The `static` member class `SortCriteria` at (11) in the non-`static` member class `SortedList` is allowed, as a `static` member type can be declared in non-static context.

Example 9.2 *Static Member Types*

Click here to view code image

```
// File: ListPool.java
package smc;

public class ListPool {                          // (1) Top-level class

  public static class MyLinkedList {             // (2) Static member class

    private interface ILink { }                  // (3) Static member interfac

    public static class BiNode                   // (4) Static member class
                  implements IBiLink {

      public static void printSimpleName() {     // (5) Static method
        System.out.println(BiNode.class.getSimpleName());
      }

      public void printName() {                  // (6) Instance method
        System.out.println(this.getClass().getName());
      }
    } // end BiNode
  } // end MyLinkedList

  interface IBiLink
            extends MyLinkedList.ILink {         // (7) Static member interfac
//  private static class Traversal { }           // (8) Compile-time error!
```

```
                                                    //     Can only be public.

      class BiTraversal { }                         // (9) Class is public and st
    } // end IBiLink

    public class SortedList {                       // (10) Non-static member cla
      private static class SortCriteria {}          // (11) Static member class
    }
  }
```

Click here to view code image

```
// File: MyBiLinkedList.java
package smc;

public class MyBiLinkedList implements ListPool.IBiLink {         // (12)

  public static void main(String[] args) {
    ListPool.MyLinkedList.BiNode.printSimpleName();               // (13)
    ListPool.MyLinkedList.BiNode node1
             = new ListPool.MyLinkedList.BiNode();                // (14)
    node1.printName();                                            // (15)

//  ListPool.MyLinkedList.ILink ref;                 // (16) Compile-time erro
  }
}
```

Output from the program:

Click here to view code image

```
BiNode
smc.ListPool$MyLinkedList$BiNode
```

Using Qualified Name of Nested Types

The *qualified name* of a (static or non-`static`) member type includes the names of the enclosing types it is lexically nested in—that is, it associates the member type with its enclosing types. In Example 9.2, the qualified name of the `static` member class `BiNode` at (4) is `ListPool.MyLinkedList.BiNode`. The qualified name of the nested interface `IBiLink` at (7) is `ListPool.IBiLink`, determined by the lexical nesting of the types. Each member class or interface is uniquely identified by this naming syntax, which is a generalization of the naming scheme for packages. The qualified name can be used in exactly the same way as any other top-level class or interface name, as shown at (12) and (13). Such a member's *fully qualified name* is its qualified name prefixed by the name of its package. For example, the fully qualified name of the `static` member class at (4) is `smc.ListPool.MyLinkedList.BiNode`. Note that a nested member type cannot have the same name as an enclosing type.

If the source file `ListPool.java` containing the declarations in

Example 9.2 is compiled, it will result in the generation of the following class files in the package `smc`, where each class file corresponds to either a class or an interface declaration in the source file:

Click here to view code image

```
ListPool$IBiLink$BiTraversal.class
ListPool$IBiLink.class
ListPool$MyLinkedList$BiNode.class
ListPool$MyLinkedList$ILink.class
ListPool$MyLinkedList.class
ListPool$SortedList.class
ListPool.class
```

Note how the full class name corresponds to the class file name (minus the extension), with the dollar symbol (`$`) replaced by the dot (`.`).

Within the scope of its top-level type, a `static` member type can be referenced regardless of its access modifier and lexical nesting, as shown at (7) in Example 9.2. Although the interface `MyLinkedList.ILink` has `private` access, it is accessible at (7), outside its enclosing class. Its access modifier (and that of the types making up its qualified name) comes into play when it is referenced by an external client. The declaration at (16) in Example 9.2 will not compile because the member interface `ListPool.MyLinkedList.ILink` has `private` access.

Instantiating Static Member Classes

A `static` member class can be instantiated without first creating an instance of the enclosing class. Example 9.2 shows a client creating an instance of a `static` member class at (14) using the `new` operator and the qualified name of the class. Not surprisingly, the compiler will flag an error if any of the types in the qualified name are not accessible by an external client.

Click here to view code image

```
ListPool.MyLinkedList.BiNode objRef1
         = new ListPool.MyLinkedList.BiNode();               // (14)
```

External clients must use the (fully) qualified name of a `static` member class in order to access such a class.

A `static` member class is loaded and initialized when the types in its enclosing context are loaded at runtime. Analogous to top-level classes, nested `static` members can always be accessed by the qualified name of the class, and no instance of the enclosing type is required, as shown at (13) where the full class name is used to invoke the `static` method `printSimpleName()` at (5) in the `static` member class `BiNode`. At (15), the reference `node1` is used to invoke the instance method `print-Name()` at (6) in an instance of the `static` member class `BiNode`. An instance of a `static` member class can exist independently of any instance of its enclosing class.

Importing Static Member Types

There is seldom any reason to import nested types from packages. It would undermine the encapsulation achieved by such types. However, a compilation unit can use the import facility to provide a shortcut for the names of member types. Note that type import and static import of `static` member types are equivalent. Type import can be used to import the `static` member type as a type name, and static import can be used to import the `static` member type as the name of a `static` member.

Usage of the (`static`) `import` declaration for `static` member classes is illustrated in Example 9.3. In the file `Client1.java`, the `import` statement at (1) allows the `static` member class `BiNode` to be referenced as `MyLinkedList.BiNode` at (2), whereas in the file `Client2.java`, the static import at (3) allows the same class to be referenced using its simple name, as at (4). At (5), the fully qualified name of the `static` member interface is used in an `implements` clause. However, in Example 9.2 at (5), the interface `smc.ListPool.IBiLink` is declared with package access in its enclosing class `ListPool` in the package `smc`, and therefore is not visible in other packages, including the default package.

Example 9.3 *Importing Static Member Types*

Click here to view code image

```
// File: Client1.java
import smc.ListPool.MyLinkedList;                            // (1) Type import

public class Client1 {
  MyLinkedList.BiNode objRef1 = new MyLinkedList.BiNode(); // (2)
}
```

Click here to view code image

```
// File: Client2.java
import static smc.ListPool.MyLinkedList.BiNode;         // (3) Static import

public class Client2 {
  BiNode objRef2 = new BiNode();                        // (4)
}

//class BiListPool implements smc.ListPool.IBiLink { }  // (5) Compile-time er
                                                        // Not accessible!
```

Accessing Members in Enclosing Context

Static member classes do not have a `this` reference, as they do not have any notion of an enclosing instance. This means that *any* code in a `static` member class can *only* directly access `static` members in its enclosing context. Trying to access any instance members directly in its enclosing context results in a compile-time error.

Figure 9.2 is a class diagram that illustrates static member classes and interfaces. These are shown as members of the enclosing context, with the `{static}` tag to indicate that they are `static` members. Since they are members of a class or an interface, their accessibility can be specified exactly like that of any other member of a class or interface: Class members can be declared with any of the four access levels (`public`, `protected`, package, `private`), but interface members are always implicitly `public`. The classes from the diagram are implemented in Example 9.4.

Figure 9.2 *Static Member Classes and Interfaces*

Example 9.4 *Accessing Members in Enclosing Context (Static Member Classes)*

Click here to view code image

```
// File: ListPool.java
public class ListPool {                                 // Top-level class

  public void messageInListPool() {                     // Instance method
    System.out.println("This is a ListPool object.");
  }

  private static class MyLinkedList {                   // (1) Static member c
    private static int maxNumOfLists = 100;             // Static field
    private int currentNumOfLists;                      // Instance field

    public static void messageInLinkedList() {          // Static method
      System.out.println("This is MyLinkedList class.");
    }

    interface ILink { int MAX_NUM_OF_NODES = 2000; }// (2) Static member inter

    protected static class Node implements ILink {  // (3) Static member class
      private int max = MAX_NUM_OF_NODES;           // (4) Instance field

      public void messageInNode() {                 // Instance method
//      int currentLists = currentNumOfLists;// (5) Not OK. Access instance fi
                                             //             in outer class
```

```
        int maxLists = maxNumOfLists;     // Access static field in outer clas
        int maxNodes = max;               // Access instance field in member c

//      messageInListPool();    // (6) Not OK. Call instance method in outer c
        messageInLinkedList();  // (7) Call static method in outer class
      }

      public static void main(String[] args) {
        int maxLists = maxNumOfLists;    // (8) Access static field in outer c
//      int maxNodes = max;   // (9) Not OK. Access instance field in member c
        messageInLinkedList();// (10) Call static method in outer class

      }
    }  // Node
  }  // MyLinkedList
} // ListPool
```

Compiling and running the program:

Click here to view code image

```
>javac ListPool.java
>java ListPool$MyLinkedList$Node
This is MyLinkedList class.
```

Example 9.4 demonstrates direct access of members in the enclosing context of the `static` member class `Node` defined at (3). The class `Node` implements the `static` member interface `ILink` declared at (2). The initialization of the instance field `max` at (4) is valid, since the field `MAX_NUM_OF_NODES`, defined in the outer interface `ILink` at (2), is implicitly `static`. The compiler will flag an error at (5) and (6) in the instance method `messageInNode()` because direct access to instance members in the enclosing context is not permitted by *any* method in a `static` member class. It will also flag an error at (9) in the method `main()` because a `static` method cannot directly access instance fields in its own class. The statements at (8) and (10) can directly access `static` members in the enclosing context. The references in these two statements can also be specified using qualified names.

Click here to view code image

```
int maxLists = ListPool.MyLinkedList.maxNumOfLists;              // (8')
ListPool.MyLinkedList.messageInLinkedList();                     // (10')
```

Note that a `static` member class can define both `static` and instance members, like any other top-level class. However, its code can only directly access `static` members in its enclosing context.

Example 9.4 also illustrates that `static` member types when declared as class members can have any access level. In the class `ListPool`, the `static` member class `MyLinkedList` at (1) has `private` access, whereas its `static` member interface `ILink` at (2) has package access and its `static` member class `Node` at (3) has `protected` access.

The class `Node` defines the method `main()` which can be executed by the following command:

Click here to view code image

```
>java ListPool$MyLinkedList$Node
```

Note that the class `Node` in the command line is specified using the qualified name of the class file minus the extension.

9.3 Non-Static Member Classes

Declaring Non-Static Member Classes

Non-`static` member classes are *inner classes*—that is, non-`static` nested classes— that are defined without the keyword `static` as instance members of either a class, an enum type, or a record class. Non-`static` member classes cannot be declared as an instance member in an interface, as a class member in an interface is implicitly `static`. Non-`static` member classes are on par with other non-`static` members defined in a reference type.

Since a non-static member class can be an instance member of a class, an enum type, or a record class, it can have any accessibility: `public`, package, `protected`, or `private`.

The compiler generates separate class files for the non-`static` member classes defined in a top-level type declaration, as it does for static member classes.

A typical application of non-`static` member classes is implementing data structures. For example, a class for linked lists could define the nodes in the list with the help of a non-`static` member class which could be declared `private` so that it was not accessible outside the top-level class, but also nodes could not exist without the list object of the enclosing class. Nesting promotes encapsulation, and the close proximity allows classes to exploit each other's capabilities.

Example 9.5 illustrates nesting and use of non-`static` member classes, and is in no way meant to be a complete implementation for linked lists. The class `MyLinkedList` at (1) defines a non-`static` member class `Node` at (5). The class `Node` has `public` access.

Example 9.5 *Defining and Instantiating Non-Static Member Classes*

Click here to view code image

```
// File: ListClient.java
class MyLinkedList {                                // (1)
  private String message = "Shine the light";       // (2)

  public Node makeNode(String info, Node next) {    // (3)
    return new Node(info, next);                    // (4)
  }

  public class Node {                               // (5) Non-static member c
    // Static field:
    static int maxNumOfNodes = 100;                 // (6)

    // Instance fields:
    private String nodeInfo;                        // (7)
    private Node next;

    // Non-zero argument constructor:
    public Node(String nodeInfo, Node next) {       // (8)
      this.nodeInfo = nodeInfo;
      this.next = next;
    }

    // Instance methods:
    public Node getNext() { return next; }
    @Override
    public String toString() {
      return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")"; // (9)
    }
  }
}
//____________________________________________________________________________
public class ListClient {                                   // (10)
  public static void main(String[] args) {                  // (11)
    MyLinkedList list = new MyLinkedList();                 // (12)
    MyLinkedList.Node node1 = list.makeNode("node1", null); // (13)
    MyLinkedList.Node node2 = list.new Node("node2", node1);// (14)
    for (MyLinkedList.Node node = node2;
         node!=null;
         node = node.getNext()) {                           // (15)
      System.out.println(node);
    }

//  MyLinkedList.Node nodeX
//               = new MyLinkedList.Node("nodeX", node1);   // (16) Not OK.

  }
}
```

Output from the program:

Click here to view code image

```
Shine the light in node1 (100)
Shine the light in node2 (100)
```

Instantiating Non-Static Member Classes

An instance of a non-`static` member class can only exist when associated with an instance of its enclosing class. This means that an instance of a non-`static` member class must be created in the context of an instance of the enclosing class. In other words, the non-`static` member class does not provide any services; only instances of the class do.

A special form of the `new` operator (called the *qualified class instance creation expression*) is used to instantiate a non-`static` member class and associate it with the immediately enclosing object:

Click here to view code image

enclosing_object_reference`.new` *non_static_member_class_constructor_call*

The *enclosing object reference* in the instance creation expression evaluates to an instance of the immediately enclosing class in which the designated non-`static` member class is defined. A new instance of the non-`static` member class is created and associated with the indicated instance of the enclosing class. Note that the expression returns a reference value that denotes a new instance of the non-`static` member class. It is illegal to specify the qualified name of the non-`static` member class in the constructor call, as the enclosing context is already given by the *enclosing object reference*.

In Example 9.5, the non-`static` method `makeNode()` at (3) in the class `MyLinkedList` illustrates how to instantiate a non-`static` member class in non-static context within the enclosing class. The non-`static` method `makeNode()` creates an instance of the non-`static` member class `Node` using the `new` operator, as shown at (4):

Click here to view code image

```
return new Node(info, next);                      // (4)
```

This creates an instance of the non-static member class `Node` in the context of the current instance of the enclosing class on which the `makeNode()` method is invoked. The `new` operator in the statement at (4) has an implicit `this` reference as the *enclosing object reference* that denotes this outer object. In the qualified class instance creation expression at (4') below, the `this` reference is explicitly specified to indicate the enclosing object:

Click here to view code image

```
return this.new Node(info, next);                 // (4')
```

The `makeNode()` method is called at (13). This method call associates an inner object of the `Node` class with the `MyLinkedList` object denoted by the reference `list`. This inner object is denoted by the reference `node1`. This reference can now be used in the normal way to access members of the inner object.

In Example 9.5, the declaration statement at (14) in the `main()` method illustrates how external clients can instantiate a non-`static` member class using the qualified class instance creation expression. The reference `list` at (12) denotes an object of the enclosing class `MyLinkedList`. This reference is specified in the qualified class instance creation expression, as shown at (14).

Click here to view code image

```
MyLinkedList.Node node2 = list.new Node("node2", node1); // (14)
```

After the execution of the statement at (14), the `MyLinkedList` object denoted by the `list` reference has two instances of the non-`static` member class `Node` associated with it. This is depicted in Figure 9.3, where the outer object (denoted by `list`) of the class `MyLinkedList` is shown with its two associated inner objects (denoted by the references `node1` and `node2`, respectively) right after the execution of the statement at (14). In other words, multiple objects of the non-`static` member classes can be associated with an object of the enclosing class at runtime.

Figure 9.3 *Outer Object with Associated Inner Objects*

In Example 9.5, if the non-`static` method `makeNode()` at (3) in the class `MyLinkedList` is made `static`, the constructor call to the `Node` class at (4) will *not* compile. Static code in a class can only refer to other `static` members, and not to non-`static` members. A `static` method would have to provide an instance of the outer object, as would any other external client, seen here in the `static` version of the `makeNode()` method:

Click here to view code image

```
public static Node makeNode(String info, Node next) {        // (3') Static met
  return new MyLinkedList().new Node(info, next);    // (4') Explicit outer obj
}
```

An example of using the inner objects is shown at (15) in the `for` loop. The print statement in the loop body calls the `toString()` method implicitly on each inner object to print its text representation.

An attempt to create an instance of the non-`static` member class using the `new` operator with the qualified name of the inner class, as shown at (16), results in a compile-time error. The full class name creation expression at (16) applies to creating instances of `static` member classes.

Accessing Members in Enclosing Context

An implicit reference to the enclosing object is always available in every method and constructor of a non-static member class. A method or constructor can explicitly specify this reference using a special form of the `this` construct and access its enclosing object, as explained in the next example.

From within a non-static member class, it is possible to refer to all members in the enclosing class directly, unless they are hidden. An example is shown at (9) in Example 9.5, where the instance field `message` in an object of the enclosing class is accessed by its simple name in the non-static member class. It is also possible to explicitly refer to members in the enclosing class, but this requires special usage of the `this` reference. One might be tempted to write the statement at (9) as follows:

Click here to view code image

```
return this.message + " in " + this.nodeInfo +
                        " (" + this.maxNumOfNodes + ")";          // (9a) Not
```

The reference `this.nodeInfo` is correct because the field `nodeInfo` certainly belongs to the current object (denoted by `this`) of the `Node` class, but `this.message` *cannot* possibly work, as the current object (indicated by `this`) of the `Node` class has no field named `message`. The correct syntax is the following:

Click here to view code image

```
return MyLinkedList.this.message + " in " + this.nodeInfo +
                                     " (" + this.maxNumOfNodes + ")"; // (9b)
```

The expression (called the *qualified* `this`)

enclosing_class_name`.this`

evaluates to a reference that denotes the enclosing object (of the specified class) that is associated with the current instance of a non-static member class.

Accessing Hidden Members

Fields and methods in the enclosing context can be *hidden* by fields and methods with the same names in the non-static member class. The qualified `this` can be used to access members in the enclosing context, somewhat analogous to using the keyword `super` in subclasses to access hidden superclass members.

Example 9.6 *Qualified* `this` *and Qualified Class Instance Creation Expression*

Click here to view code image

```
// File: OuterInstances.java
class TLClass {                                            // (1)  TLC
  private String id = "TLClass ";                          // (2)
  public TLClass(String objId) { id = id + objId; }        // (3)
  public void printId() {                                  // (4)
    System.out.println(id);
  }

  class InnerB {                                           // (5)  NSMC
    private String id = "InnerB ";                         // (6)
    public InnerB(String objId) { id = id + objId; }       // (7)
    public void printId() {                                // (8)
      System.out.print(TLClass.this.id + " : ");           // (9)  Refers to (
      System.out.println(id);                              // (10) Refers to (
    }

    class InnerC {                                         // (11) NSMC
      private String id = "InnerC ";                       // (12)
      public InnerC(String objId) { id = id + objId; }     // (13)
      public void printId() {                              // (14)
        System.out.print(TLClass.this.id + " : ");         // (15) Refers to (
        System.out.print(InnerB.this.id + " : ");          // (16) Refers to (
        System.out.println(id);                            // (17) Refers to (
      }
      public void printIndividualIds() {                   // (18)
        TLClass.this.printId();                            // (19) Calls (4)
        InnerB.this.printId();                             // (20) Calls (8)
        printId();                                         // (21) Calls (14)
      }
    } // InnerC
  } // InnerB
} // TLClass
//____________________________________________________________________________
public class OuterInstances {                                // (22)
  public static void main(String[] args) {                   // (23)
    TLClass a = new TLClass("a");                            // (24)
    TLClass.InnerB b = a.new InnerB("b");                    // (25) b  --> a
    TLClass.InnerB.InnerC c1 = b.new InnerC("c1");           // (26) c1 --> b
    TLClass.InnerB.InnerC c2 = b.new InnerC("c2");           // (27) c2 --> b
    b.printId();                                             // (28)
    c1.printId();                                            // (29)
    c2.printId();                                            // (30)
    System.out.println("------------");

    TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
    TLClass.InnerB.InnerC cc = bb.new InnerC("cc");          // (32)
    bb.printId();                                            // (33)
```

```
      cc.printId();                                         // (34)
      System.out.println("------------");

      TLClass.InnerB.InnerC ccc =
        new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
      ccc.printId();                                        // (36)
      System.out.println("------------");

      ccc.printIndividualIds();                             // (37)
    }
  }
```

Output from the program:

Click here to view code image

```
TLClass a : InnerB b
TLClass a : InnerB b : InnerC c1
TLClass a : InnerB b : InnerC c2
------------
TLClass aa : InnerB bb
TLClass aa : InnerB bb : InnerC cc
------------
TLClass aaa : InnerB bbb : InnerC ccc
------------
TLClass aaa
TLClass aaa : InnerB bbb
TLClass aaa : InnerB bbb : InnerC ccc
```

Example 9.6 illustrates the qualified `this` employed to access members in the enclosing context, and also demonstrates the qualified class instance creation expression employed to create instances of non-static member classes. The example shows the non-static member class `InnerC` at (11), which is nested in the non-static member class `InnerB` at (5), which in turn is nested in the top-level class `TLClass` at (1). All three classes have a `private` non-static `String` field named `id` and a non-`static` method named `printId`. The member name in the nested class *hides* the name in the enclosing context. These members are *not* overridden in the nested classes because no inheritance is involved. In order to refer to the hidden members, the nested class can use the qualified `this`, as shown at (9), (15), (16), (19), and (20).

Within the nested class `InnerC`, the three forms used in the following statements to access its field `id` are equivalent:

Click here to view code image

```
System.out.println(id);                     // (17)
System.out.println(this.id);                // (17a)
System.out.println(InnerC.this.id);         // (17b)
```

The `main()` method at (23) uses the special syntax of the `new` operator to create objects of non-static member classes and associate them with enclosing objects. An instance of class `InnerC` (denoted by `c1`) is created at (26) in the context of an instance of class `InnerB` (denoted by `b`), which was created at (25) in the context of an instance of class `TLClass` (denoted by `a`), which in turn was created at (24).

Click here to view code image

```
TLClass a = new TLClass("a");                              // (24)
TLClass.InnerB b = a.new InnerB("b");                      // (25) b  --> a
TLClass.InnerB.InnerC c1 = b.new InnerC("c1");             // (26) c1 --> b
```

The reference `c1` is used at (29) to invoke the method `printId()` declared at (14) in the nested class `InnerC`. This method prints the field `id` from all the objects associated with an instance of the nested class `InnerC`.

Click here to view code image

```
TLClass a : InnerB b : InnerC c1
```

When the intervening references to an instance of a non-static member class are of no interest—that is, if the reference values need not be stored in variables—the `new` operator can be chained as shown at (31) and (35).

Click here to view code image

```
TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
...
TLClass.InnerB.InnerC ccc =
  new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
```

Note that the (outer) objects associated with the instances denoted by the references `c1`, `cc`, and `ccc` (at (26), (32), and 35), respectively) are distinct, as evident from the program output. However, the instances denoted by references `c1` and `c2` (at (26) and (27), respectively) have the same outer objects associated with them.

Inheritance Hierarchy and Enclosing Context

A non-`static` member class can extend another class and implement interfaces, as any normal class. An inherited field (or method) in a non-`static` member subclass can *hide* a field (or method) with the same name in the enclosing context. Using the simple name to access this member will access the inherited member, not the one in the enclosing context.

Example 9.7 illustrates the situation. In the inner subclass at (4), the field name `value` at (1) in the superclass hides the field with the same name in the enclosing class at (3). In Example 9.7, the standard form of the `this` reference is used to access the inherited field `value`, as shown at (6). The simple name of the field would also work in this case, as would the keyword `super` with the simple name. The `super` keyword would be mandatory to access the superclass field if the inner subclass also declared a field with the same name. However, to access the member from the enclosing context, the qualified `this` must be used, as shown at (7).

Example 9.7 *Inheritance Hierarchy and Enclosing Context*

Click here to view code image

```
// File: HiddenAndInheritedAccess.java
class Superclass {
  protected int value = 3;                      // (1) Instance field in superclass
}
//_____________________________________________________________________________
class TopLevelClass {                           // (2) Top-level Class
  private double value = 3.14;                  // (3) Hidden by the instance fiel

                                                //     at (1) in the inner subclas
  class InnerSubclass extends Superclass {      // (4) Non-static member subclass
    public void printHidden() {                 // (5)
      // (6) value from superclass:
      System.out.println("this.value: " + this.value);

      // (7) value from enclosing context:
      System.out.println("TopLevelClass.this.value: "
                         + TopLevelClass.this.value);
    }
  } // InnerSubclass
} // TopLevelClass
//_____________________________________________________________________________
public class HiddenAndInheritedAccess {
  public static void main(String[] args) {
    TopLevelClass.InnerSubclass ref = new TopLevelClass().new InnerSubclass();
    ref.printHidden();
  }
}
```

Output from the program:

Click here to view code image

```
this.value: 3
TopLevelClass.this.value: 3.14
```

Some caution should be exercised when extending an inner class. Some of the subtleties involved are illustrated by Example 9.8. The nesting and the inheritance hierarchy of the classes involved are shown in Figure 9.4. The question that arises is how do we provide an *outer instance* when creating a *subclass instance* of a non-static member class—for example, when creating objects of the subclasses `SubInnerA` and `InnerB` in Figure 9.4.

Figure 9.4 *Non-Static Member Classes and Inheritance*

Example 9.8 *Extending Inner Classes*

Click here to view code image

```
// File: Extending.java
class OuterA {                                   // (1)
  class InnerA { }                               // (2)
}
//____________________________________________________________________________
class SubInnerA extends OuterA.InnerA {          // (3) Extends NSMC at (2)

  // (4) Mandatory non-zero argument constructor:
  SubInnerA(OuterA outerRef) {
    outerRef.super();                            // (5) Explicit super() call
  }
}
//____________________________________________________________________________
class OuterB extends OuterA {                    // (6) Extends class at (1)
  class InnerB extends OuterA.InnerA { }         // (7) Extends NSMC at (2)
}
//____________________________________________________________________________
public class Extending {
  public static void main(String[] args) {

    // (8) Outer instance passed explicitly in constructor call:
    SubInnerA obj1 = new SubInnerA(new OuterA());
    System.out.println(obj1.getClass());

    // (9) No outer instance passed explicitly in constructor call to InnerB:
    OuterB.InnerB obj2 = new OuterB().new InnerB();
    System.out.println(obj2.getClass());
  }
}
```

Output from the program:

```
class SubInnerA
class OuterB$InnerB
```

In Example 9.8, the non-static member class `InnerA`, declared at (2) in the class `OuterA` at (1), is extended by `SubInnerA` at (3). Note that `SubInnerA` and the class `OuterA` are not related in any way, and that the subclass `OuterB` inherits the class `InnerA` from its superclass `OuterA`. An instance of `SubInnerA` is created at (8). An instance of the class `OuterA` is explicitly passed as an argument in the constructor call to `SubInnerA`. The constructor at (4) for `SubInnerA` has a special `super()` call in its body at (5), called a *qualified superclass constructor invocation*. This call ensures that the constructor of the superclass `InnerA` has an outer object (denoted by the reference `outerRef`) to bind to. Using the standard `super()` call in the subclass constructor is not adequate because it does not provide an outer instance for the superclass constructor to bind to. The non-zero argument constructor at (4) and the `outerRef.super()` expression at (5) are mandatory to set up the relationships correctly between the objects involved.

The outer object problem mentioned above does not arise if the subclass that extends an inner class is also declared within an outer class that extends the outer class of the superclass. This situation is illustrated at (6) and (7): The classes `InnerB` and `OuterB` extend the classes `InnerA` and `OuterA`, respectively. The member class `InnerA` is inherited by the class `OuterB` from its superclass `OuterA`—and can be regarded as being nested in the class `OuterB`. Thus an object of class `OuterB` can act as an outer object for both an instance of class `InnerA` and that of class `InnerB`. The object creation expression `new OuterB().new InnerB()` at (9) creates an `OuterB` object and implicitly passes its reference to the default constructor of class `InnerB`. The default constructor of class `InnerB` invokes the default constructor of its superclass `InnerA` by calling `super()` and passing it the reference of the `OuterB` object, which the constructor of class `InnerA` can readily bind to.

It goes without saying that such convoluted inheritance and nesting relationships as those in Example 9.8 hardly qualify as best coding practices.

Review Questions

9.1 Which statement is true about the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    Outer objRef = new Outer();
    System.out.println(objRef.createInner().getSecret());
  }
}

class Outer {
  private int secret;
  Outer() { secret = 123; }

  class Inner {
    int getSecret() { return secret; }
  }
  Inner createInner() { return new Inner(); }
}
```

Select the one correct answer.

- **a.** The program will fail to compile because the class `Inner` cannot be declared within the class `Outer`.
- **b.** The program will fail to compile because the method `createInner()` cannot be allowed to pass objects of the class `Inner` to methods outside the class `Outer`.
- **c.** The program will fail to compile because the field `secret` is not accessible from the method `getSecret()`.
- **d.** The program will fail to compile because the method `getSecret()` is not accessible from the `main()` method in the class `MyClass`.
- **e.** The code will compile and print `123` at runtime.

9.2 Which of the following statements are true about nested classes? Select the two correct answers.

- **a.** An instance of a static member class has an implicit outer instance.
- **b.** A static member class can contain non-`static` fields.
- **c.** A static member interface can contain non-`static` fields.
- **d.** A static member interface has an implicit outer instance.
- **e.** An instance of the outer class can be associated with many instances of a non-static member class.

9.3 Which statement is true about the following program?

Click here to view code image

```
public class Nesting {
  public static void main(String[] args) {
    B.C obj = new B().new C();
  }
}

class A {
  int val;
  A(int v) { val = v; }
}

class B extends A {
  int val = 1;
  B() { super(2); }

  class C extends A {
    int val = 3;
    C() {
      super(4);
      System.out.println(B.this.val);
      System.out.println(C.this.val);
      System.out.println(super.val);
    }
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.
b. The program will compile and print `2`, `3`, and `4`, in that order at runtime.
c. The program will compile and print `1`, `4`, and `2`, in that order at runtime.
d. The program will compile and print `1`, `3`, and `4`, in that order at runtime.
e. The program will compile and print `3`, `2`, and `1`, in that order at runtime.

9.4 Which of the following statements are true about the following program?

Click here to view code image

```
public class Outer {
  public void doIt() {}
  public class Inner {
    public void doIt() {}
  }

  public static void main(String[] args) {
    new Outer().new Inner().doIt();
  }
}
```

Select the two correct answers.

a. The `doIt()` method in the `Inner` class overrides the `doIt()` method in the `Outer` class.
b. The `doIt()` method in the `Inner` class overloads the `doIt()` method in the `Outer` class.
c. The `doIt()` method in the `Inner` class hides the `doIt()` method in the `Outer` class.
d. The qualified name of the `Inner` class is `Outer.Inner`.
e. The program will fail to compile.

9.4 Local Classes

Declaring Local Classes

A local class is an inner class that is defined in a block. This can be essentially any context where a local block or block body is allowed: a method, a constructor, an initializer block, a `try-catch-finally` construct, loop bodies, or an `if-else` statement. Example 9.9 shows declaration of the local class `StaticLocal` at (5) that is defined in the static context of the method `staticMethod()` at (1).

A local class cannot have any access modifier and cannot be declared `static`, as shown at (4) in Example 9.9. However, it can be declared `abstract` or `final`, as shown at (5). The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations. In particular, it must be declared before use in the block. In Example 9.9, an attempt to create an object of class `StaticLocal` at (2) and use the class `Static-Local` at (3) fails, as the class has not been defined before use, but this is not a problem at (11), (12), (13), and (14).

A local class can declare members and constructors, shown from (6) to (10), as in a normal class. The members of the local class can have any access level, and are accessible in the enclosing block regardless of their access level. Even though the field `if1` at (7) is `private`, it is accessible in the enclosing method at (12).

Blocks in non-static context have a `this` reference available, which refers to an instance of the class containing the block. An instance of a local class, which is declared in such a non-`static` block, has an instance of the enclosing class associated with it. This gives such a non-`static` local class much of the same capability as a non-static member class.

However, if the block containing a local class declaration is defined in static context (i.e., a `static` method or a static initializer block), the local class is implicitly `static` in the sense that its instantiation does not require any outer object. This aspect of local classes is reminiscent of static member classes. However, note that a local class cannot be specified with the keyword `static`. The `static` method at (1) is called at (15). The local class `StaticLocal` can only be instantiated, as shown at (11), in the enclosing method `staticMethod()` and does not require any outer object of the enclosing class. Analogous to the value of a local variable, the object of the local class is not available to the caller of the method after the method completes execution, unless measures are taken to store it externally or if its reference value is returned by the call.

Example 9.9 *Declaring Local Classes*

Click here to view code image

```
// File: LocalClient1.java
class TLCWithSLClass {                                 // Top-level Class

  static void staticMethod(final int fp) {             // (1) Static Method
//  StaticLocal slRef = new StaticLocal(10);           // (2) Class cannot be reso
//  System.out.println(StaticLocal.staticValue()); // (3) Class cannot be reso

//  public static class StaticLocal { // (4) Not OK. Cannot be static,
                                      //              and no access modifier
    final class StaticLocal {                            // (5) Static local class
      public static final int sf1 = 10;                  // (6) Static field
      private int if1;                                   // (7) Instance field
      public StaticLocal(int val) {                      // (8) Constructor
        this.if1 = val;
      }
      public int getValue() { return if1; }              // (9) Instance method
      public static int staticValue() { return sf1; }// (10) Static method
    } // end StaticLocal

    StaticLocal slRef2 = new StaticLocal(100);                             //
    System.out.println("Instance field: " + slRef2.if1);                   //
    System.out.println("Instance method call: " + slRef2.getValue());      //
    System.out.println("Static method call: " + StaticLocal.staticValue());//
  } // end staticMethod
}

public class LocalClient1 {
  public static void main(String[] args) {
    TLCWithSLClass.staticMethod(100);                                      //
  }
}
```

Output from the program:

```
Instance field: 100
Instance method call: 100
Static method call: 10
```

Accessing Declarations in Enclosing Context

Declaring a local class in a static or a non-static block influences what the class can access in the enclosing context.

Accessing Local Declarations in the Enclosing Block

Example 9.10 illustrates how a local class can access declarations in its enclosing block. Example 9.10 shows declaration of the local class `NonStaticLocal` at (7) that is defined in the non-static context of the method `nonStaticMethod()` at (1).

A local class can access variables (local variables, method parameters, and `catch`-block parameters) that are declared `final` or *effectively* `final` in the scope of its local context. A variable whose value does not change after it is initialized is said to be *effectively* `final`. This situation is shown at (8) and (9) in the `NonStaticLocal` class, where the `final` parameter `fp` and the effectively `final` local variable `flv` of the method `nonStaticMethod()` are accessed. Access to local variables that are not `final` or effectively `final` is not permitted from local classes. The local variable `nfv1` at (4) is accessed at (10) in the local class, but this local variable is not effectively `final` as it is reassigned a new value at (6).

Accessing a local variable from the local context that has not been declared or has not been *definitely assigned* (§5.5, p. 232) results in a compile-time error, as shown at (11) and (12). The local variable `nflv2` accessed at (11) is not declared before use, as it is declared at (16). The local variable `nflv3` accessed at (12) is not initialized before use, as it is initialized at (17)—which means it is not definitely assigned at (12).

Declarations in the local class can *shadow* declarations in the enclosing block. The field `hlv` at (13) shadows the local variable by the same name at (3) in the enclosing method. There is no way for the local class to refer to shadowed declarations in the enclosing block.

The non-`static` method at (1) is called at (19) on an instance of its enclosing class. When the constructor at (15) in the non-`static` method is executed, the reference to this instance is passed implicitly to the constructor, thus this instance acts as the enclosing object of the local class instance.

Example 9.10 *Accessing Local Declarations in the Enclosing Block (Local Classes)*

Click here to view code image

```
// File: LocalClient2.java
class TLCWithNSLClass {                // Top-level Class

  void nonStaticMethod(final int fp) { // (1) Non-static Method
    // Local variables:
    int flv = 10;              // (2) Effectively final local variable
    final int hlv = 20;        // (3) Final local variable (constant variable)
    int nflv1 = 30;            // (4) Non-final local variable
    int nflv3;                 // (5) Non-final local variable declaration

    nflv1 = 40;                // (6) Not effectively final local variable

    // Non-static local class
    class NonStaticLocal {// (7)
      int f1 = fp;        // (8) Final param from enclosing method
      int f2 = flv;       // (9) Effectively final variable from enclosing met
//    int f3 = nflv1;     // (10) Not effectively final from enclosing method
//    int f4 = nflv2;     // (11) Name nflv2 cannot be resolved: use-before-de
//    int f5 = nflv3;     // (12) Not definitely assigned
      int hlv;            // (13) Shadows local variable at (3)
      NonStaticLocal (int value) {
        hlv = value;
        System.out.println("Instance field: " + hlv);// (14) Prints value from
      }
    } // end NonStaticLocal

    NonStaticLocal nslRef = new NonStaticLocal(200);// (15) Implicit outer obj
    int nflv2 = 50;                                 // (16) Attempted use in (
    nflv3 = 60;                                     // (17) Initializes (4)
    System.out.println("Local variable: " + hlv);   // (18) Prints value from
  } // end nonStaticMethod
}

public class LocalClient2 {
  public static void main(String[] args) {
    new TLCWithNSLClass().nonStaticMethod(1000);    // (19)
  }
}
```

Output from the program:

```
Instance field: 200
Local variable: 20
```

Accessing Members in the Enclosing Class

Example 9.11 illustrates how a local class can access members in its enclosing class. The top-level class `TLCWith2LCS` declares two methods: `nonStaticMethod()` and `staticMethod()`. Both methods define a local class each: `NonStaticLocal` at (1) in non-static context and `StaticLocal` at (8) in static context, both of which are subclasses of the superclass `Base`.

A local class can access members inherited from its superclass in the usual way. The field `nsf1` in the superclass `Base` is inherited by the local subclass `NonStaticLocal`. This inherited field is accessed in the `NonStaticLocal` class, as shown at (2), (3), and (4), by using the field's simple name, the standard `this` reference, and the `super` keyword, respectively. This also applies for static local classes, as shown at (9), (10), and (11).

Fields and methods in the enclosing class can be *hidden* by member declarations in the local class. The non-`static` field `nsf1`, inherited by the local classes, hides the field by the same name in the enclosing class `TLCWith2LCS`. The qualified `this` can be used in non-`static` local classes for *explicit* referencing of members in the enclosing class, regardless of whether these members are hidden or not.

Click here to view code image

```
double f4 = TLCWith2LCS.this.nsf1;      // (5) In enclosing object.
```

However, the special form of the `this` construct *cannot* be used in a local class that is declared in static context, as shown at (12), since it does not have any notion of an outer object. A local class in static context cannot refer to non-static members in the enclosing context.

A non-`static` local class can access both `static` and non-`static` members defined in the enclosing class. The non-`static` field `nsf2` and `static` field `sf` are defined in the enclosing class `TLCWith2LCS`. They are accessed in the `NonStaticLocal` class at (6) and (7), respectively. The special form of the `this` construct can also be used in non-`static` local classes, as previously mentioned.

However, a local class that is declared in static context can only directly access `static` members defined in the enclosing class. The `static` field `sf` in the class `TLCWith2LCS` is accessed in the `StaticLocal` class at (14), but the non-`static` field `nsf1` cannot be accessed, as shown at (13).

Example 9.11 *Accessing Members in the Enclosing Class (Local Classes)*

Click here to view code image

```
// File: LocalClient3.java
class Base { protected int nsf1; }     // Superclass
//______________________________________________________________________
class TLCWith2LCS {                    // Top-level Class
  private int nsf1;                    // Non-static field
  private int nsf2;                    // Non-static field
  private static int sf;               // Static field

  void nonStaticMethod( int fp) {      // Non-static Method

    class NonStaticLocal extends Base {// (1) Non-static local subclass
      int f1 = nsf1;                   // (2) Inherited from superclass.
      int f2 = this.nsf1;              // (3) Inherited from superclass.
      int f3 = super.nsf1;             // (4) Inherited from superclass.
      int f4 = TLCWith2LCS.this.nsf1;  // (5) In enclosing object.
      int f5 = nsf2;                   // (6) Instance field in enclosing obje
      int f6 = sf;                     // (7) static field from enclosing clas
    } // NonStaticLocal

  } // nonStaticMethod

  static void staticMethod(final int fp) { // Static Method

    class StaticLocal extends Base {   // (8) Static local subclass
      int f1 = nsf1;                   // (9) Inherited from superclass.
      int f2 = this.nsf1;              // (10) Inherited from superclass.
      int f3 = super.nsf1;             // (11) Inherited from superclass.
//    int f4 = TLCWith2LCS.this.nsf1;  // (12) No enclosing object.
//    int f5 = nsf2;                   // (13) No enclosing object.
      int f6 = sf;                     // (14) static field from enclosing cla
    } // StaticLocal

  } // staticMethod
}

public class LocalClient3 {
  public static void main(String[] args) {
    TLCWith2LCS.staticMethod(200);          // (15)
    new TLCWith2LCS().nonStaticMethod(100); // (16)
  }
}
```

Instantiating Local Classes

Clients outside the scope of a local class cannot instantiate the class directly because such classes are, after all, local. A local class can be instantiated in the block in which it is defined. Like a local variable, a local class must be declared before being used in the block.

A method can return instances of any local class it declares. The local class type must then be assignable to the return type of the method. The return type cannot be the same as the local class type, since this type is not accessible outside the method. A supertype of the local class must be specified as the return type. This also means that, in order for the objects of the local class to be useful outside the method, a local class should implement an interface or override the behavior of its supertypes.

Example 9.12 illustrates how clients can instantiate local classes. The nesting and the inheritance hierarchy of the classes involved are shown in Figure 9.5. The non-`static` local class `Circle` at (5) is defined in the non-`static` method `createCircle()` at (4), which has the return type `Shape`. The static local class `Graph` at (9) is defined in the `static` method `createGraph()` at (8), which has the return type `IDrawable`.

Figure 9.5 *Local Classes and Inheritance Hierarchy*

Example 9.12 *Instantiating Local Classes*

Click here to view code image

```
// File: LocalClassClient.java
interface IDrawable {                        // (1)
  void draw();
}
//____________________________________________________________________
class Shape implements IDrawable {           // (2)
  @Override
  public void draw() { System.out.println("Drawing a Shape."); }
}
//____________________________________________________________________
```

```
class Painter {                           // (3) Top-level Class
  public Shape createCircle(final double radius) { // (4) Non-static Method
    class Circle extends Shape {          // (5) Non-static local class
      @Override
      public void draw() {
        System.out.println("Drawing a Circle of radius: " + radius); // (6)
      }
    }
    return new Circle();                  // (7) Passed enclosing object refer
  }

  public static IDrawable createGraph() { // (8) Static Method
    class Graph implements IDrawable {    // (9) Static local class
      @Override
      public void draw() { System.out.println("Drawing a Graph."); }
    }
    return new Graph();                   // (10) Object of static local class
  }
}
//___________________________________________________________________________
public class LocalClassClient {
  public static void main(String[] args) {
    IDrawable[] drawables = {             // (11)
        new Painter().createCircle(5),    // (12) Object of non-static local c
        Painter.createGraph(),            // (13) Object of static local class
        new Painter().createGraph()       // (14) Object of static local class
    };
    for (IDrawable aDrawable : drawables) // (15)
      aDrawable.draw();

    System.out.println("Local Class Names:");
    System.out.println(drawables[0].getClass().getName());   // (16)
    System.out.println(drawables[1].getClass().getName());   // (17)
  }
}
```

Output from the program:

Click here to view code image

```
Drawing a Circle of radius: 5.0
Drawing a Graph.
Drawing a Graph.
Local Class Names:
Painter$1$Circle
Painter$1$Graph
```

The `main()` method in Example 9.12 creates a polymorphic array `drawables` of type `IDrawable[]` at (11), which is initialized at (12) through (14) with instances of the local classes.

Creating an instance of a non-`static` local class requires an instance of the enclosing class. In Example 9.12, the non-`static` method `createCircle()` is invoked on the instance of the enclosing class `Painter` to create an instance of the non-`static` local class `Circle` in the non-`static` method, as shown at (12). The reference to the instance of the enclosing class is passed implicitly in the constructor call at (7) to the non-`static` local class.

A `static` method can be invoked either through the class name or through a reference of the class type. An instance of a static local class can be created either way by calling the `createGraph()` method, as shown at (13) and (14). As might be expected in static context, no outer object is involved.

As references to a local class cannot be declared outside the local context, the functionality of the class is only available through supertype references. The method `draw()` is invoked on objects in the array `IDrawable` at (15). The program output indicates which objects were created. In particular, note that the `final` parameter `radius` of the method `createCircle()` at (4) is accessed in the `draw()` method of the local class `Circle` at (6). An instance of the local class `Circle` is created at (12) by a call to the method `createCircle()`. The `draw()` method is invoked on this instance of the local class `Circle` in the loop at (15). The value of the `final` parameter `radius` is still accessible to the `draw()` method invoked on this instance, although the call to the method `createCircle()`, which created the instance in the first place, has completed. Values of (effectively) `final` local variables continue to be available to instances of local classes whenever these values are needed.

The output in Example 9.12 also shows the actual names of the local classes. In fact, the local class names are reflected in the generated class file names. Because multiple local class declarations with the same name can be defined in the methods of the enclosing class, a numbering scheme (`$i`) is used to generate distinct class file names.

9.5 Static Local Types

It is possible to declare local interfaces, local enum types, and local record classes. However, these local nested types are implicitly `static`—as opposed to local classes that are never `static`. A local class declared in a static context is *not* the same as a static local type, and they are compared next.

Since these local nested types are implicitly `static`, this has implications for these types, as listed below. We illustrate these implications for a static local record class in Example 9.13, but they apply equally to local interfaces and local enum types as well.

- A local record class can be instantiated with the `new` operator without specifying an immediately enclosing instance. Interfaces and enum types cannot be instantiated with the `new` operator.

At (10) in Example 9.13, an instance of the static local record class is created without passing a reference to the enclosing object.

- Static local types can only access `static` members in the enclosing context—this is the same as local classes declared in static context, whereas local classes declared in non-static context can access all members in the enclosing context.

  In Example 9.13, it is possible to access the `static` field in the enclosing class in the declaration of the local record class at (7) and (9), but it is not possible to access non-static fields as shown at (8).

- Static local types cannot access any local variables in the enclosing method—in contrast to local classes that can access (effectively) `final` local variables in the enclosing method.

  In Example 9.13, it is not possible to access any local variables, as shown at (5) and (6), in the enclosing method.

Example 9.13 *Defining Static Local Record Classes*

Click here to view code image

```
// File: LocalTypesClient.java
class LocalTypes {                     // Top-level Class
  private int nsf;                     // (1) Non-static field
  private static int sf;               // (2) Static field

  void nonStaticMethod(final int fp) { // (3) Non-static Method. Final paramet
    int lv = 20;                       // (4) Local variable

    record StaticLocalRecord(int val) { // Static local record
      // Cannot access local variables:
//    static int f1 = fp;  // (5) Cannot access final param from enclosing met
//    static int f2 = lv;  // (6) Cannot access effectively final local variab
                           //     from enclosing method.

      // Can only access static fields in enclosing context:
      static int f3 = sf;         // (7) Access static field in enclosing cont

      void printFieldsFromEnclosingContext() {
//      System.out.println(nsf);  // (8) Cannot access non-static field
                                  //     in enclosing context.
        System.out.println(sf);   // (9) Access static field in enclosing cont
      }
    }
    // (10) Create local record. No enclosing instance passed to the construct
    StaticLocalRecord lrRef = new StaticLocalRecord(100);
    System.out.println("Value: " + lrRef.val());
  } // nonStaticMethod
}

public class LocalTypesClient {
  public static void main(String[] args) {
    new LocalTypes().nonStaticMethod(1000);    // (10)
  }
}
```

Output from the program:

```
Value: 100
```

9.6 Anonymous Classes

Declaring Anonymous Classes

Classes are usually first defined and then instantiated using the `new` operator. Anonymous classes combine the process of definition and instantiation into a single step. Anonymous classes are defined at the location they are instantiated, using additional syntax with the `new` operator. As these classes do not have a name, an instance of the class can only be created together with the definition. Like local classes, anonymous classes are inner classes that can be defined in static and non-static context.

An anonymous class can be defined and instantiated in contexts where a reference value can be used—that is, as expressions that evaluate to a reference value denoting an object of the anonymous class. Anonymous classes are typically used for creating objects *on the fly* in contexts such as the value in a `return` statement, an argument in a method call, or in initialization of variables. The reference value of an anonymous class object can be assigned to any kind of variable (fields and local variables) whose type is a supertype of the anonymous class.

An anonymous class cannot be declared with an access modifier, nor can it be declared `static`, `final`, or `abstract`.

Typical uses of anonymous classes are to implement *event listeners* in GUI-based applications, threads for simple tasks (see examples in Chapter 22, p. 1365), and comparators for providing a total ordering for objects (see Example 14.11, p. 772).

Extending an Existing Class

The following syntax can be used for defining and instantiating an anonymous class that extends an existing class specified by *superclass name*:

Click here to view code image

```
new superclass_name<optional_type_arguments>  (optional_constructor_arguments)
{
 member_declarations
}
```

Optional type arguments and constructor arguments can be specified, which are passed to the superclass constructor. Thus the superclass must provide a constructor corresponding to the arguments passed. No `extends` clause is used in the construct. Since an anonymous class cannot define constructors (as it does not have a name), an instance initializer can be used to achieve the same effect as a no-arg constructor.

Both static and non-static members can be declared in the class body. An anonymous class can override any instance methods accessible by their simple name from the superclass, but if it extends an `abstract` class, then it must provide implementation for all `abstract` methods from the superclass. The declaration is terminated by a semicolon (`;`), unless the reference value of the resulting object is immediately used to access a member of this object.

Example 9.14 *Defining Anonymous Classes*

Click here to view code image

```
// File: AnonClassClient.java
interface IDrawable {                                   // (1)
  void draw();
}
//______________________________________________________________________
class Shape implements IDrawable {                      // (2)
  @Override
  public void draw() { System.out.println("Drawing a Shape."); }
}
//______________________________________________________________________
class Painter {                                         // (3) Top-level Class

  public Shape createShape() {                          // (4) Non-static Method
    return new Shape() {                                // (5) Extends superclass at (
      @Override
      public void draw() { System.out.println("Drawing a new Shape."); }
    };
  }
  public static IDrawable createIDrawable() {   // (7) Static Method
    return new IDrawable() {                    // (8) Implements interface at
      @Override
      public void draw() {
        System.out.println("Drawing a new IDrawable.");
      }
    };
  }
}
//____________________________________________________________________________
public class AnonClassClient {
  public static void main(String[] args) {      // (9)
    IDrawable[] drawables = {                   // (10)
        new Painter().createShape(),            // (11) Non-static anonymous c
        Painter.createIDrawable(),              // (12) Static anonymous class
        new Painter().createIDrawable()         // (13) Static anonymous class
    };
    for (IDrawable aDrawable : drawables)       // (14)
      aDrawable.draw();

    System.out.println("Anonymous Class Names:");
    System.out.println(drawables[0].getClass().getName());// (15)
    System.out.println(drawables[1].getClass().getName());// (16)
  }
}
```

Output from the program:

```
Drawing a new Shape.
Drawing a new IDrawable.
Drawing a new IDrawable.
Anonymous Class Names:
Painter$1
Painter$2
```

Class declarations from Example 9.12 are adapted to use anonymous classes in Example 9.14. The non-`static` method `createShape()` at (4) defines a non-`static` anonymous class at (5), which extends the superclass `Shape`. The anonymous class at (5) overrides the inherited method `draw()` from the superclass `Shape` at (2).

Click here to view code image

```
// ...
class Shape implements IDrawable {          // (2)
  @Override public void draw() { System.out.println("Drawing a Shape."); }
}

class Painter {                             // (3) Top-level Class

  public Shape createShape() {              // (4) Non-static Method
    return new Shape() {                    // (5) Extends superclass at (2)
      @Override public void draw() { System.out.println("Drawing a new Shape."
    };
  }
  // ...
}
// ...
```

Implementing an Interface

The following syntax can be used for defining and instantiating an anonymous class that implements an interface specified by the *interface name*:

Click here to view code image

```
new interface_name<optional_parameterized_types>() { member_declarations }
```

An anonymous class provides a *single* interface implementation. No arguments are passed, as an interface does not define a constructor. The anonymous class implicitly extends the `Object` class. Note that no `implements` clause is used in the construct. The class body must provide implementation for all `abstract` methods declared in the interface.

An anonymous class implementing an interface is shown below. Details can be found in Example 9.14. The `static` method `createIDrawable()` at (7) defines a static anonymous class at (8), which implements the interface `IDrawable`, by providing an implementation of the method `draw()`. The functionality of objects of an anonymous class that implements an interface is available through references of the interface type and the `Object` type—that is, its supertypes.

Click here to view code image

```
interface IDrawable {                          // (1) Interface
  void draw();
}
// ...
class Painter {                                // (3) Top-level Class
  // ...
  public static IDrawable createIDrawable() {  // (7) Static Method
    return new IDrawable() {                   // (8) Implements interface at
      @Override public void draw() {
        System.out.println("Drawing a new IDrawable.");
      }
    };
  }
}
// ...
```

The following code is an example of a typical use of anonymous classes in building GUI applications. The anonymous class at (1) implements the `java.awt.event.-ActionListener` interface that has the method `actionPerformed()`. When the `addActionListener()` method is called on the GUI button denoted by the reference `quitButton`, the anonymous class is instantiated and the reference value of the object is passed as a parameter to the method. The method `addActionListener()` of the GUI button registers the reference value, and when the user clicks the GUI button, it can invoke the `actionPerformed()` method on the `ActionListener` object.

Click here to view code image

```
quitButton.addActionListener(
    new ActionListener() {    // (1) Anonymous class implements an interface
      // Invoked when the user clicks the quit button.
      @Override public void actionPerformed(ActionEvent evt) {
        System.exit(0);       // (2) Terminates the program
      }
    }
);
```

Instantiating Anonymous Classes

The discussion on instantiating local classes (see Example 9.12) is also valid for instantiating anonymous classes. The class `AnonClassClient` in Example 9.14 creates one instance at (11) of the non-`static` anonymous class defined at (5) by calling the non-`static` method `createShape()` on an instance of the class `Painter`, and two instances at (12) and (13) of the anonymous class that is defined in static context at (8) by calling the `static` method `createDrawable()` in the class `Painter`. The program output shows the polymorphic behavior and the runtime types of the objects. Similar to a non-static local class, an instance of a non-`static` anonymous class has an instance of its enclosing class associated with it. An enclosing instance is not mandatory for creating objects of a `static` anonymous class.

The names of the anonymous classes at runtime are also shown in the program output in Example 9.14. A numbering scheme (`$i`) is used to designate the anonymous classes according to their declaration order inside the enclosing class. They are also the names used to designate their respective class files. Anonymous classes are not so anonymous after all.

Referencing Instances of an Anonymous Class

Each time an anonymous class declaration is executed it returns a reference value of a new instance of the anonymous class. Either an anonymous class extends an existing class or it implements an interface. Usually it makes sense to override (and implement) methods from its supertype. References of its supertype can refer to objects of the anonymous subclass. As we cannot declare references of an anonymous class, an instance of an anonymous class can only be manipulated by a supertype reference. Any subclass-specific members in an anonymous class cannot be accessed directly by external clients—the only functionality available is that provided by inherited methods and overridden methods which can be invoked by a supertype reference denoting an anonymous subclass instance.

Accessing Declarations in Enclosing Context

Member declarations and access rules for local classes (p. 512) also apply to anonymous classes. Example 9.15 is an adaptation of Example 9.9, Example 9.10, and Example 9.11. It illustrates what members can be declared in an anonymous class and what can be accessed in its enclosing context. The local classes from previous examples have been adapted to anonymous classes in Example 9.15.

The `TLCWithAnonClasses` class declares a non-`static` method at (3) in which a local variable (`baseRef`) is initialized with an instance of a non-`static` anonymous class at (9). The `baseRef` variable is used to invoke the `printValue()` method on this instance at (28).

The `TLCWithAnonClasses` class declares a `static` field `baseField` at (29) that is initialized with an instance of a `static` anonymous class. The field `baseField` is used to invoke the `printValue()` method on this instance at (36).

Accessing Local Declarations in the Enclosing Block

Being inner classes, there are many similarities between local classes and anonymous classes. An anonymous class can only access (effectively) `final` variables in its enclosing local context, shown both for static fields at (10), (11), and (12), and for instance fields at (13), (14), and (15). Local variables accessed in the anonymous class must be declared before use, and definitely assigned, as shown at (16) and (17), respectively. A field name in the anonymous class can shadow a local variable with the same name in the local context, as shown at (18). Note that the anonymous class declared at (29) does *not* have an enclosing local context, only an enclosing class, as it is defined in non-static context.

Accessing Members in the Enclosing Class

A member (either inherited or declared) in an anonymous class can hide a member with the same name in the enclosing object or class.

The inherited field `nsf1` hides the field by the same name in the enclosing class, as shown at (19), (20), and (21).

Non-static anonymous classes can access any non-hidden members in the enclosing class by their simple names and the qualified `this`, as shown at (22), (23), and (24). Static anonymous classes can only access non-hidden `static` members in the enclosing class by their simple names, as shown at (30), (31), and (32).

Example 9.15 *Accessing Declarations in Enclosing Context (Anonymous Classes)*

Click here to view code image

```
// File: AnonClient.java
abstract class Base {           // (1) Superclass
  protected int nsf1;
  abstract void printValue();
}
//____________________________________________________________________________
class TLCWithAnonClasses {      // (2) Top level Class
  private int nsf1;             // Non-static field
  private int nsf2;             // Non-static field
  private static int sf = 5;    // Static field

  public void nonStaticMethod(final int fp) { // (3) Non-static Method
    // Local variables:
    int flv = 10;               // (4) Effectively final local variable
    final int hlv = 20;         // (5) Final local variable (constant variable
    int nflv1 = 30;             // (6) Non-final local variable
    int nflv3;                  // (7) Non-final local variable declaration

    nflv1 = 40;                 // (8) Not effectively final local variable

    Base baseRef = new Base() { // (9) Non-static anonymous class
      // Static fields: Accessing local declarations in the enclosing block:
      static int sff1 = fp;   // (10) Final param from enclosing method
      static int sff2 = flv;  // (11) Effect. final variable from enclosing me
//    static int sf1 = nflv1; // (12) Not effect. final from enclosing method

      // Instance fields: Accessing local declarations in the enclosing block:
      int f1 = fp;       // (13) Final param from enclosing method
      int f2 = flv;      // (14) Effectively final variable from enclosing met
//    int f3 = nflv1;    // (15) Not effectively final from enclosing method

//    int f4 = nflv2;    // (16) nflv2 cannot be resolved: not decl-before-use
//    int f5 = nflv3;    // (17) Not definitely assigned: not initialized
      int hlv;           // (18) Shadows local variable at (5)

      // Accessing member declarations inherited from superclass:
      int f6 = nsf1;                   // (19) Inherited from superclass
      int f7 = this.nsf1;              // (20) Inherited from superclass
      int f8 = super.nsf1;             // (21) Inherited from superclass

      // Accessing (hidden) member declarations in the enclosing class:
      int f9 = TLCWithAnonClasses.this.nsf1;          // (22) In enclosing obj
      int f10 = nsf2;                  // (23) Instance field in enclosing obj
      int f11 = sf;                    // (24) Static field from enclosing cla
      { nsf1 = fp; }                   // (25) Non-static initializer block

      @Override void printValue() {                        // (26) Instance me
        System.out.println("Instance field nsf1: " + nsf1);// (27)
      }
    };

    int nflv2 = 70;
    nflv3 = 80;
    baseRef.printValue();              // (28) Invoke method on anonymous obje
  }

  public static final Base baseField = new Base() { // (29) Static anonymous c
    // Accessing (hidden) member declarations in the enclosing class:
//  int f1 = TLCWithAnonClasses.this.nsf1; // (30) Not OK. No enclosing object
//  int f2 = nsf2;                         // (31) Not OK. No enclosing object
    { nsf1 = sf; }                         // (32) Non-static initializer bloc

    @Override void printValue() {                        // (33) Instance meth
      System.out.println("Instance field nsf1: " + nsf1);// (34)
    }
  };
}
//____________________________________________________________________________
public class AnonClient {
  public static void main(String[] args) {
    new TLCWithAnonClasses().nonStaticMethod(100);                // (35)
    TLCWithAnonClasses.baseField.printValue();                    // (36)
  }
}
```

Output from the program:

Click here to view code image

```
Instance field nsf1: 100
Instance field nsf1: 5
```

Review Questions

9.5 Which statement is true about nested classes?

Select the one correct answer.

a. Non-static member classes must have either package or `public` access.

b. All nested classes cannot declare static member classes.

c. Methods in all nested classes cannot be declared `static`.

d. All nested classes can be declared `static`.

e. Static member classes can declare non-`static` methods.

9.6 Given the declaration:

Click here to view code image

```
interface IntHolder { int getInt(); }
```

Which of the following methods are valid?

Click here to view code image

```
//----(1)----
  IntHolder makeIntHolder(int i) {
    i = 10;
    return new IntHolder() {
      public int getInt() { return i; }
    };
  }
//----(2)----
  IntHolder makeIntHolder(final int i) {
    return new IntHolder {
      public int getInt() { return i; }
    };
  }
//----(3)----
  IntHolder makeIntHolder(int i) {
    class MyIH implements IntHolder {
      public int getInt() { return i; }
    }
    return new MyIH();
  }
//----(4)----
  IntHolder makeIntHolder(final int i) {
    class MyIH implements IntHolder {
      public int getInt() { return i; }
    }
    return new MyIH();
  }
//----(5)----
  IntHolder makeIntHolder(int i) {
    return new MyIH(i);
  }
  static class MyIH implements IntHolder {
    final int j;
    MyIH(int i) { j = i; }
    public int getInt() { return j; }
  }
```

Select the three correct answers.

a. The method at (1).

b. The method at (2).

c. The method at (3).

d. The method at (4).

e. The method at (5).

9.7 Which statement is true about nested classes?

Select the one correct answer.

a. No other static members, except `static final` fields declared as constant variables, can be declared within a non-static member class.

b. If a non-static member class is nested within a class named

`Outer`, methods within the non-static member class must use the prefix `Outer.this` to access the members of the class `Outer`.

c. All fields in any nested class must be declared `final`.

d. Anonymous classes cannot have constructors.

e. If the reference `objRef` denotes an instance of any nested class within the class `Outer`, the expression `(objRef instanceof Outer)` will evaluate to `true`.

9.8 Which expression can be inserted independently at (1) so that compiling and running the program will print `LocalVar.str1`?

Click here to view code image

```
public class Access {
  final String str1 = "Access.str1";

  public static void main(final String args[]) {
    final String str1 = "LocalVar.str1";

    class Helper { String getStr1() { return str1; } }
    class Inner {
      String str1 = "Inner.str1";
      Inner() {
        System.out.println( /* (1) INSERT EXPRESSION HERE */ );
      }
    }
    Inner inner = new Inner();
  }
}
```

Select the one correct answer.

a. `str1`

b. `this.str1`

c. `Access.this.str1`

d. `new Helper().getStr1()`

e. `this.new Helper().getStr1()`

f. `Access.new Helper().getStr1()`

g. `new Access.Helper().getStr1()`

h. `new Access().new Helper().getStr1()`

9.9 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  private char x = '=';
  public static void main(String[] args) {
    char x = '<';
    Test t = new Test() {
      private char x = '>';
      public String toString() {
        return this.x + super.toString() + x;
      }
    };
    System.out.println(t);
  }
  public String toString() {
      return x + "42";
    }
  }
```

Select the one correct answer.

a. The program will print `<=42>`.
b. The program will print `<=42<`.
c. The program will print `>=42>`.
d. The program will print `>=42<`.
e. The program will compile, but it will throw an exception at runtime.
f. The program will fail to compile.

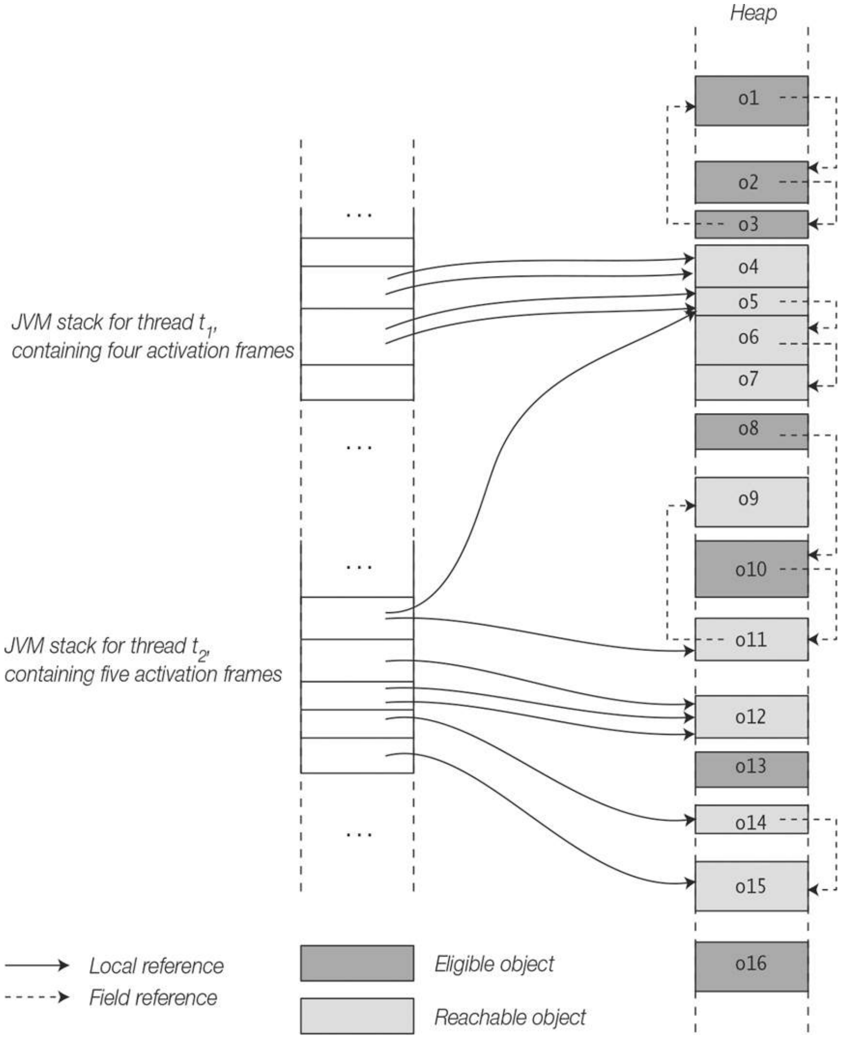
9.10 Which statement is true about the following program?

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    int x = 42;
    String s = new String() {
      int x = 24;
      public int hashCode() {
        return this.x;
      }
    };
    System.out.println(s.hashCode());
  }
}
```

Select the one correct answer.

a. The program will print `42`.
b. The program will print `24`.
c. The program will compile, but it will throw an exception at runtime.
d. The program will fail to compile.


Heap
o1
o2
o3
o4
o5
o6
o7
o8
o9
o10
o11
o12
o13
o14
o15
o16
...
JVM stack for thread t_1, containing four activation frames
JVM stack for thread t_2, containing five activation frames
Local reference
Field reference
Eligible object
Reachable object

Object Lifetime 10

Chapter Topics

- Understanding automatic garbage collection and guidelines for facilitating garbage collection
- Using static and instance initializers, in particular declaration order, forward references, and exception handling
- Understanding the role played by initializers and constructors in initializing objects, classes, and interfaces

Java SE 17 Developer Exam O	
§10.1, p. 533 to §10.4, p. 537	[3.1] Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection ❍ *Object lifecycle and garbage collection are covered in this chapter.* ❍ *For constructing objects, see Chapter 3, p. 97.* ❍ *For nested classes, see Chapter 9, p. 489.*
§10.5, p. 540 to §10.9, p. 555	[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ❍ *Initializers and object construction are covered in this chapter.* ❍ *For declaring normal classes and defining class members and constructors, see Chapter 3, p. 97.* ❍ *For records, see §5.14, p. 299.*

Java SE 11 Developer Exam O	
§10.1, p. 533 to §10.4, p. 537	[3.1] Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection) ❍ *Object lifecycle and garbage collection are covered in this chapter.* ❍ *For constructing objects, see Chapter 3, p. 97.* ❍ *For nested classes, see Chapter 9, p. 489.*

Java SE 11 Developer Exam O	
§10.5, p. 540 to	[3.3] Initialize objects and their members using instance and static initialiser statements and constructors

§10.9, p. 555

- *Initializers and object construction are covered in this chapter.*
- *For initializing objects using initialization statements and constructors, see Chapter 3, p. 97.*

One of the most important features of Java is its *automatic memory management*, also know as *automatic garbage collection*. This chapter provides a non-technical discussion of this important topic about reclaiming storage of unused objects in a program at runtime. In addition, we look at the role of initializer expressions, initializer blocks, and constructors to initialize the state of an object when it is created with the `new` operator, and which includes the construction of the state from its superclasses.

10.1 Garbage Collection

Efficient memory management is essential in a runtime system. Storage for objects is allocated in a designated part of the memory called the *heap*, which has a finite size. Garbage collection (GC) is a process of managing the heap efficiently, by reclaiming memory occupied by objects that are no longer needed and making it available for new objects. Java provides automatic garbage collection, meaning that the runtime environment can take care of memory management without the program having to take any special action. Objects allocated on the heap (through the `new` operator) are administered by the automatic garbage collector. The automatic garbage collection scheme guarantees that a reference to an object is always valid while the object is needed by the program. Specifically, the object will not be reclaimed if it will result in a *dangling reference*—that is, a reference to an object that no longer exists.

Having an automatic garbage collector frees the programmer from the responsibility of writing code for deleting objects. By relying on the automatic garbage collector, a Java program also forfeits any significant influence on the garbage collection of its objects (p. 537). However, this price is insignificant when compared to the cost of putting the code for object management in place and plugging all the memory leaks. Time-critical applications should recognize that the automatic garbage collector runs as a background task and may have a negative impact on their performance.

10.2 Reachable Objects

An automatic garbage collector essentially performs two tasks:

- Decides if and when memory needs to be reclaimed
- Finds objects that are no longer needed by the program and reclaims their storage

A program has no guarantees that the automatic garbage collector will be run during its execution. Consequently, a program should not rely on the scheduling of the automatic garbage collector for its behavior (p. 537).

To understand how the automatic garbage collector finds objects whose storage should be reclaimed, we need to look at the activity happening in the JVM. Java provides *thread-based multitasking*, meaning that several threads can be executing concurrently in the JVM, each doing its own task. A *thread* is an independent path of execution through the program code. A thread is alive if it has not completed its execution. Each live thread has its own JVM stack (§7.1, p. 365). The JVM stack contains activation frames of methods that are currently active. Local references declared in a method can always be found in the method's activation frame, stored on the JVM stack associated with the thread in which the method is called. Objects, in contrast, are always created on the heap. If an object has a field reference, the field will be found inside the object in the heap, and the object denoted by the field reference will also be found in the heap.

An example of how memory is organized during execution is depicted in Figure 10.1, which shows two live threads (t_1 and t_2) and their respective JVM stacks with the activation frames. The diagram indicates which objects in the heap are referenced by local references in the method activation frames. It also identifies field references in objects, which refer to other objects in the heap. Some objects have several aliases.

An object in the heap is said to be *reachable* if it is referenced by any *local* reference in a JVM stack. Likewise, any object that is denoted by a reference in a reachable object is said to be reachable. Reachability is a transitive relationship. Thus a reachable object has at least one chain of reachable references from the JVM stack. Any reference that makes an object reachable is called a *reachable reference*. An object that is not reachable is said to be *unreachable*.

A reachable object is *alive*, and is *accessible* by a live thread. Note that an object can be accessible by more than one thread. Any object that is *not* accessible by a live thread is a candidate for garbage collection. When an object becomes unreachable and is waiting for its memory to be reclaimed, it is said to be *eligible* for garbage collection. An object is eligible for garbage collection if all references denoting it are in eligible objects. Eligible objects do not affect the future course of program execution. When the garbage collector runs, it finds and reclaims the storage of eligible objects, although garbage collection does not necessarily occur as soon as an object becomes unreachable.

In Figure 10.1, the objects `o4`, `o5`, `o11`, `o12`, `o14`, and `o15` all have reachable references. Objects `o13` and `o16` have no reachable references, and therefore, are eligible for garbage collection.

From the preceding discussion we can conclude that if a compound object becomes unreachable, its constituent objects also become unreachable, barring any reachable references to the constituent objects. Although the objects `o1`, `o2`, and `o3` in Figure 10.1 form a circular list, they do not have any reachable references. Thus these objects are all eligible for garbage collection. Conversely, the objects `o5`, `o6`, and `o7` form a linear list, but they are all reachable, as the first object in the list, `o5`, is reachable. The objects `o8`, `o10`, `o11`, and `o9` also form a linear list (in that order), but not all objects in the list are reachable. Only the objects `o9` and `o11` are reachable, as object `o11` has a reachable reference. The objects `o8` and `o10` are eligible for garbage collection.

The *lifetime* of an object is the time from its creation to the time it is garbage collected. Under normal circumstances, an object is accessible from the time when it is created to the time when it becomes unreachable. The lifetime of an object can also include a period when it is eligible for garbage collection, waiting for its storage to be reclaimed.

Figure 10.1 *Memory Organization at Runtime*

10.3 Facilitating Garbage Collection

The automatic garbage collector determines which objects are not reachable, and therefore, are eligible for garbage collection. It will certainly go to work if there is an imminent memory shortage. Even so, automatic garbage collection should not be perceived as a license for creating a plethora of objects and then forgetting about them. Nevertheless, certain programming practices can help in minimizing the overhead associated with garbage collection during program execution.

Certain objects, such as files and network connections, can tie up resources and should be disposed of properly when they are no longer needed. In most cases, the `try`-with-resources statement (§7.7, p. 407) provides a convenient facility for such purposes, as it will always ensure proper closing of any `AutoCloseable` resources.

To optimize its memory footprint, a live thread should retain access to an object for only as long as the object is needed for its execution. The program can allow objects to become eligible for garbage collection as early as possible by removing all references to an object when it is no longer needed.

Objects that are created and accessed by local references in a method are eligible for garbage collection when the method terminates, unless reference values to these objects are exported out of the method. This can occur if a reference value is returned from the method, passed as an argument to another method that records the reference value, or thrown as an exception. However, a method need not always leave objects to be garbage collected after its termination. It can facilitate garbage collection by taking suitable action—for example, by nulling references.

Click here to view code image

```
import java.io.*;

class WellBehavedClass {
  // ...
  void wellBehavedMethod() {
    FileReader reader;
    long[] bigArray = new long[20000];
    // ... uses local variables ...
    // Does clean-up (before starting something extensive)
    reader = null;                      // (1)
    bigArray = null;                    // (2)

    // Start some other extensive activity
    // ...
  }
  // ...
}
```

In this code, the local variables are set to `null` after use at (1) and (2), before starting some other extensive activity. This makes the objects denoted by the local variables eligible for garbage collection from this point onward, rather than after the method terminates. This optimization technique of nulling references should be used only as a last resort when resources are scarce.

Here are some other techniques to facilitate garbage collection:

- When a method returns a reference value and the object denoted by the value is not needed, not assigning this value to a reference facilitates garbage collection.
- If a reference is assigned a new value, the object that was previously denoted by the reference can become eligible for garbage collection.
- Removing reachable references to a compound object can make the constituent objects become eligible for garbage collection, as explained earlier.

10.4 Invoking Garbage Collection Programmatically

Although Java provides facilities to invoke the garbage collector explicitly, there are no guarantees that it will be run. The program can request that garbage collection be performed, but there is no way to force garbage collection to be activated.

The `System.gc()` method can be used to request garbage collection.

> `static void gc()`
>
> Requests that garbage collection be run.

Alternatively, corresponding methods in the `Runtime` class can be used. A Java application has a unique `Runtime` object that can be used by the application to interact with the JVM. An application can obtain this object by calling the method `Runtime.getRuntime()`. The `Runtime` class provides several methods related to memory issues:

> `static Runtime getRuntime()`
>
> Returns the `Runtime` object associated with the current application.
>
> `void gc()`
>
> Requests that garbage collection be run. There are no guarantees that it will be run. It is recommended to use the more convenient `static` method `System.gc()`.

> `long freeMemory()`
>
> Returns the amount of free memory (bytes) in the JVM that is available for new objects.
>
> `long totalMemory()`
>
> Returns the total amount of memory (bytes) available in the JVM, including both memory occupied by current objects and memory available for new objects.

The following points regarding automatic garbage collection should be noted:

- Trying to initiate garbage collection programmatically does not guarantee that it will actually be run.
- Garbage collection might not even be run if the program execution does not warrant it. Thus any memory allocated during program execution might remain allocated after program termination, but will eventually be reclaimed by the operating system.
- There are also no guarantees about the order in which the objects will be garbage collected. Therefore, the program should not make any assumptions based on this criteria.
- Garbage collection does not guarantee that there will be enough memory for the program to run. A program can rely on the garbage collector to run when memory gets very low, and it can expect an `OutOfMemoryError` to be thrown if its memory demands cannot be met.

Review Questions

10.1 Which of the following statements is true?

Select the one correct answer.

a. Objects can be explicitly destroyed using the keyword `delete`.

b. An object will be garbage collected immediately after it becomes unreachable.

c. If object `obj1` is accessible from object `obj2`, and object `obj2` is accessible from `obj1`, then `obj1` and `obj2` are not eligible for garbage collection.

d. If object `obj1` can access object `obj2` that is eligible for garbage collection, then `obj1` is also eligible for garbage collection.

10.2 Identify the location in the following program where the object, initially referenced by `arg1`, is eligible for garbage collection.

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    String msg;
    String pre = "This program was called with ";
    String post = " as first argument.";
    String arg1 = new String((args.length > 0) ? "'" + args[0] + "'" :
                              "<no argument>");
    msg = arg1;
    arg1 = null;             // (1)
    msg = pre + msg + post;  // (2)
    pre = null;              // (3)
    System.out.println(msg);
    msg = null;              // (4)
    post = null;             // (5)
    args = null;             // (6)
  }
}
```

Select the one correct answer.

a. After the line at (1).
b. After the line at (2).
c. After the line at (3).
d. After the line at (4).
e. After the line at (5).
f. After the line at (6).

10.3 How many objects are eligible for garbage collection when control reaches (1)?

Click here to view code image

```
public class Elements {
  public static void main(String[] args) {
    int[] array = new int[4];
    for (int i = 0; i < 4; i++) {
      array[i] = i;
    }
    array[0] = array[1] = array[2] = array[3] = 0;
    System.gc();                          // (1);
  }
}
```

Select the one correct answer.

a. 0

b. 1

c. 4

d. It’s hard to say.

10.4 How many objects are eligible for garbage collection when control reaches (1)?

Click here to view code image

```
public class Link {
  private Link next;
  Link(Link next) { this.next = next; }

  public static void main(String[] args) {
    Link p = null;
    for (int i = 0; i < 5; i++) {
      p = new Link(p);
    }
    System.gc();                          // (1)
  }
}
```

Select the one correct answer.

a. 0

b. 5

c. 10

d. It’s hard to say.

10.5 How many objects are eligible for garbage collection when control reaches (1)?

```
public class Song {
```

Click here to view code image

```
public class Album {
  private Song[] songs = { new Song(), new Song() };
  public Song[] getSongs() {
    return songs;
  }

  public void removeAll() {
    songs = null;
  }
}
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Album album = new Album();
    Song[] songs = album.getSongs();
    album.removeAll();
    System.out.println(album.getSongs());
    // Line (1)
  }
}
```

Select the one correct answer.

a. 0
b. 1
c. 2
d. 3
e. 4

10.5 Initializers

Initializers can be used to set initial values for fields in objects and classes. There are three kinds of initializers:

- *Field initializer expressions*
- *Static initializer blocks*
- *Instance initializer blocks*

Subsequent sections in this chapter provide details on these initializers, concluding with a discussion of the procedure involved in constructing the state of an object when the object is created using the `new` operator.

For brevity, we have dispensed with the access modifier for fields in most class declarations in the rest of this chapter, as field accessibility by clients is not of primary concern here.

10.6 Field Initializer Expressions

Initialization of fields can be specified in field declaration statements using initializer expressions. The value of the initializer expression must be assignment compatible with the declared field. We distinguish between static and non-static field initializers.

Click here to view code image

```
class ConstantInitializers {
         int minAge = 12;              // (1) Non-static

  static double pensionPoints = 10.5; // (2) Static
  // ...
}
```

The fields of an object are initialized with the values of initializer expressions when the object is created by using the `new` operator. In the previous example, the declaration at (1) will result in the field `minAge` being initialized to `12` in every object of the class

`ConstantInitializers` created with the `new` operator. If no explicit initializer expressions are specified, default values are assigned to the fields.

When a class is loaded, it is initialized, meaning its static fields are initialized with the values of the initializer expressions. The declaration at (2) will result in the static field `pensionPoints` being initialized to `10.5` when the class is loaded by the JVM. Again, if no explicit initializers are specified, default values are assigned to the static fields.

An initializer expression for a static field cannot refer to non-static members by their simple names. The keywords `this` and `super` cannot occur in a static initializer expression.

Since a class is always initialized before it can be instantiated, an instance initializer expression can always refer to any static member of a class, regardless of the member declaration order. In the following code, the instance initializer expression at (1) refers to the static field `NO_OF_WEEKS` declared and initialized at (2). Such a *forward reference* is legal. More examples of forward references are given in the next subsection.

Click here to view code image

```
class MoreInitializers {
           int noOfDays     = 7 * NO_OF_WEEKS;     // (1) Non-static
  static int NO_OF_WEEKS = 52;                     // (2) Static
  // ...
}
```

Initializer expressions can also be used to define constants in interfaces (§5.6, p.254). Such initializer expressions are implicitly static, as they define values of `static final` fields in an interface.

Initializer expressions are used to initialize local variables as well (§3.4, p.102). A local variable is initialized with the value of the initializer expression every time the local variable declaration is executed.

Declaration Order of Initializer Expressions

When an object is created using the `new` operator, instance initializer expressions are executed in the order in which the instance fields are declared in the class.

Java requires that the declaration of a field must occur *before its usage* in any initializer expression if the field is *used on the right-*

hand side of an assignment in the initializer expression. This essentially means that the declaration of a field must occur before the value of the field is *read* in an initializer expression. Using the field on the left-hand side of an assignment in the initializer expression does not violate the *declaration-before-reading rule*, as this constitutes a *write* operation. This rule applies when the usage of the field is by its *simple name*.

There is one caveat to the declaration-before-reading rule: It does not apply if the initializer expression defines an anonymous class, as the usage then occurs in a different class that has its own accessibility rules in the enclosing context. The restrictions outlined earlier help to detect initialization anomalies at compile time.

In the next code example, the initialization at (2) generates a compile-time error because the field `width` in the initializer expression violates the declaration-before-reading rule. Because the usage of the field `width` in the initializer expression at (2) does not occur on the left-hand side of the assignment, this is an illegal forward reference. To remedy the error, the declaration of the field `width` at (4) can be moved in front of the declaration at (2). In any case, we can use the keyword `this` as shown at (3), but it will read the default value `0` in the field `width`.

Click here to view code image

```
class NonStaticInitializers {
  int length  = 10;                   // (1)
//double area = length * width;       // (2) Not OK. Illegal forward reference
  double area = length * this.width;  // (3) OK, but width has default value 0
  int width   = 10;                   // (4)

  int sqSide = height = 20;           // (5) OK. Legal forward reference.
  int height;                         // (6)
}
```

The forward reference at (5) is legal. The usage of the field `height` in the initializer expression at (5) occurs on the left-hand side of the assignment. The initializer expression at (5) is evaluated as `(sqSide = (height = 20))`. Every object of the class `NonStaticInitializers` will have the fields `height` and `sqSide` set to the value `20`.

The declaration-before-reading rule is equally applicable to static initializer expressions when static fields are referenced by their simple names.

Example 10.1 shows why the order of field initializer expressions can be important. The initializer expressions in this example are calls to methods defined in the class, and methods are not subject to the same access rules as initializer expressions. The call at (2) to the method `initMaxGuests()` defined at (4) is expected to return the maximum number of guests, but the field `occupancyPerRoom` at (3) will not have been explicitly initialized at this point; therefore, its default value `0` will be used in the method `initMaxGuests()`, which will return a logically incorrect value. The program output shows that after object creation, the occupancy per room is correct, but the maximum number of guests is wrong.

Example 10.1 *Initializer Expression Order and Method Calls*

Click here to view code image

```
// File: TestOrder.java
class Hotel {
  private int noOfRooms        = 12;                                 // (1)
  private int maxNoOfGuests    = initMaxGuests();                    // (2) Bu
  private int occupancyPerRoom = 2;                                  // (3)
  public int initMaxGuests() {                                       // (4)
    System.out.println("occupancyPerRoom: " + occupancyPerRoom);
    System.out.println("maxNoOfGuests:    " + noOfRooms * occupancyPerRoom);
    return noOfRooms * occupancyPerRoom;
  }

  public int getMaxGuests() { return maxNoOfGuests; }                // (5)

  public int getOccupancy() { return occupancyPerRoom; }             // (6)
}
//________________________________________________________________________
public class TestOrder {
  public static void main(String[] args) {
    Hotel hotel = new Hotel();                                       // (7)
    System.out.println("AFTER OBJECT CREATION");
    System.out.println("occupancyPerRoom: " + hotel.getOccupancy()); // (8)
    System.out.println("maxNoOfGuests:    " + hotel.getMaxGuests()); // (9)
  }
}
```

Output from the program:

```
occupancyPerRoom: 0
maxNoOfGuests:    0
AFTER OBJECT CREATION
occupancyPerRoom: 2
maxNoOfGuests:    0
```

Exception Handling and Initializer Expressions

Initializer expressions in named classes and interfaces must not result in any uncaught checked exception (§7.2, p.374). If any checked exception is thrown during execution of an initializer expression, it must be caught and handled by code called from the initializer expression. This restriction does not apply to instance initializer expressions in anonymous classes.

Example 10.2 illustrates exception handling for initializer expressions in named classes. The static initializer expression at (3) calls the static method `createHotel-Pool()` at (4), which catches and handles the checked `TooManyHotelsException` defined at (2). If the method `createHotelPool()` were to use the `throws` clause to specify the checked exception, instead of catching and handling it within a `try-catch` block, the initializer expression at (3), which called the method, would have to handle the exception. However, the initializer expression cannot specify any exception handling, as the compiler would complain.

The instance initializer expression at (5) calls the method `initMaxGuests()` at (6), which can throw the unchecked `RoomOccupancyTooHighException`. If thrown, this exception will be caught and handled in the `main()` method. Program output confirms that an unchecked `RoomOccupancyTooHighException` was thrown during program execution.

Example 10.2 *Exceptions in Initializer Expressions*

Click here to view code image

```
// File: ExceptionsInInitializers.java
class RoomOccupancyTooHighException
      extends RuntimeException {}                     // (1) Unchecked Excepti
class TooManyHotelsException
      extends Exception {}                            // (2) Checked Exception
//____________________________________________________________________________
class Hotel {
  // Static Members
  private static int noOfHotels = 12;
  private static Hotel[] hotelPool = createHotelPool();   // (3)

  private static Hotel[] createHotelPool() {              // (4)
    try {
      if (noOfHotels > 10)
        throw new TooManyHotelsException();
    } catch (TooManyHotelsException e) {
      noOfHotels = 10;
      System.out.println("No. of hotels adjusted to " + noOfHotels);
    }
    return new Hotel[noOfHotels];
  }
  // Instance Members
  private int noOfRooms        = 215;
  private int occupancyPerRoom = 5;
  private int maxNoOfGuests    = initMaxGuests();         // (5)

  private int initMaxGuests() {                           // (6)
    if (occupancyPerRoom > 4)
      throw new RoomOccupancyTooHighException();
    return noOfRooms * occupancyPerRoom;
  }
}
//____________________________________________________________________________
public class ExceptionsInInitializers {
  public static void main(String[] args) {
    try { new Hotel(); }
    catch (RoomOccupancyTooHighException exception) {
      exception.printStackTrace();
    }
  }
}
```

Output from the program:

Click here to view code image

```
No. of hotels adjusted to 10
RoomOccupancyTooHighException
      at Hotel.initMaxGuests(ExceptionsInInitializers.java:29)
      at Hotel.<init>(ExceptionsInInitializers.java:25)
      at ExceptionsInInitializers.main(ExceptionsInInitializers.java:36)
```

10.7 Static Initializer Blocks

Java allows static initializer blocks to be defined in a class. Although such blocks can include arbitrary code, they are primarily used for initializing static fields. The code in a static initializer block is executed only once, when the class is loaded and initialized.

Local variables in static and instance initializer blocks can be declared with the reserved type name `var`—that is, local variable type inference using `var` is permitted for local variable declarations in initializer blocks. Code in the rest of this chapter shows many examples of such declarations.

The syntax of a static initializer block comprises the keyword `static` followed by a local block that can contain arbitrary code, as shown at (3) in the declaration of the following class:

Click here to view code image

```
class MatrixData {

  static final int ROWS = 12, COLUMNS = 10;          // (1)
  static long[][] matrix = new long[ROWS][COLUMNS];  // (2)
  // ...
  static {                                           // (3) Static initializer
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = 2*i + j;
  }
  // ...
}
```

When the class `MatrixData` is first loaded, the `static final` fields at (1) are initialized. Then the array of arrays `matrix` of specified size is created at (2), followed by the execution of the static block at (3).

Note that the static initializer block is not contained in any method. A class can have more than one static initializer block. Initializer blocks are *not* members of a class, and they cannot have a `return` statement because they cannot be called directly.

When a class is initialized, the initializer expressions in `static` field declarations and `static` initializer blocks are executed in the order in which they are specified in the class. In the class `MatrixData`, the initializer expressions at (1) and (2) are executed before the static initializer block at (3).

Similar restrictions apply to static initializer blocks as for static initializer expressions: The keywords `this` and `super` cannot occur in a static initializer block, as such a block defines a static context.

Declaration Order of Static Initializers

In the class `ScheduleV1` below, the `static` field declaration at (1) has a *forward reference* to the `static` field `numOfWeeks` which has not been declared yet. The simple name of the `static` field `numOfWeeks` cannot be used in the initializer expression at (1) before its declaration.

Click here to view code image

```
public class ScheduleV1 {
//static int numOfDays  = 7 * numOfWeeks; // (1) Compile-time error! Simple na
  static int numOfWeeks = 52;             // (2)
}
```

The code will compile if we change the order of the declarations, and the `static` fields `numOfWeeks` and `numOfDays` will be initialized correctly with the values `52` and `364`, respectively.

Click here to view code image

```
public class ScheduleV2 {
  static int numOfWeeks = 52;                         // (2) 52
  static int numOfDays  = 7 * numOfWeeks;             // (1) 364
}
```

The code will also compile if the *class name* is used to access the field, but the `static` field `numOfDays` will *not* be initialized correctly.

Click here to view code image

```
public class ScheduleV3 {
  static int numOfDays  = 7 * ScheduleV3.numOfWeeks;  // (1) 0
  static int numOfWeeks = 52;                         // (2) 52
}
```

The code above is actually executed as follows, with the default value `0` of the `static` field `numOfWeeks` being used in the initializer expression at (1):

Click here to view code image

```
public class ScheduleV3 {
  static int numOfDays;                               // Default value: 0
  static int numOfWeeks;                              // Default value: 0
  static {
    numOfDays  = 7 * numOfWeeks;                      // (1) 0
    numOfWeeks = 52;                                  // (2) 52
  }
}
```

However, the `static` field `numOfDays` will be initialized correctly if the `static` field `numOfWeeks` is declared `final`.

Click here to view code image

```
public class ScheduleV4 {
  static int numOfDays  = 7 * ScheduleV4.numOfWeeks;  // (1) 364
  static final int numOfWeeks = 52;                   // (2) final: 52
}
```

The initializer expression `52` for the `static` field `numOfWeeks` at (2) is a *constant expression* of type `int`, which is an `int` literal. The compiler is able to compute the value of a constant expression. A `final` variable of either a primitive type or type `String` that is initialized with a constant expression is called a *constant variable*—that is, once initialized, the value of a `final` variable cannot be changed (§5.5, p. 230). At (2), the `final static` field `numOfWeeks` is a constant variable. During class initialization, such `final static` fields are always initialized first, before any other initializers are executed. Such a constant field never gets initialized with the default value of its type at runtime. Rearrangement of code done by the compiler in this case is equivalent to the following code:

Click here to view code image

```
public class ScheduleV4 {
  static final int numOfWeeks = 52;                   // Constant variable: 52
  static int numOfDays;                               // Default value: 0
  static {
    numOfDays  = 7 * numOfWeeks;                      // (1) 364
  }
}
```

Static fields should be accessed statically—that is, using the class name—which is the best policy, but care should be exercised in the order of their declaration and initialization.

When making forward references using *simple names*, code in a `static` initializer is subject to the declaration-before-reading rule. Note that this rule applies only when the use of the field is by its simple name. Using the class name to access a `static` field is never a problem.

Example 10.3 illustrates forward references and the order of execution for `static` initializer expressions in field declarations and code in `static` initializer blocks. An illegal forward reference occurs at (4), where an attempt is made to read the value of the field `sf1` before its declaration. At (11) the read operation is after the declaration, and therefore, allowed. Forward reference made on the left-hand side of the assignment is always allowed, as shown at (2), (5), and (7). The initializers are executed in their declaration order. A `static` field has the value it was last assigned in an initializer. If there is no explicit assignment, the `static` field has the default value of its type. Referring to a `static` field using the class name is always allowed.

Declaring local variables using the reserved word `var` in static initializer blocks can be found at (5) and (12) in Example 10.3.

Example 10.3 *Static Initializers and Forward References*

Click here to view code image

```
package refs1;

public class ForwardRefs {

  static {                    // (1) Static initializer block
    System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 1: sf1=0, sf2

    sf1 = 10;                 // (2) OK. Assignment to sf1 allowed
//  sf1 = if1;           // (3) Not OK. Non-static field access in static cont
//  int a = 2 * sf1;     // (4) Not OK. Read operation before declaration
    var b = sf1 = 20;         // (5) OK. Assignment to sf1 allowed
    int c = ForwardRefs.sf1;  // (6) OK. Not accessed by simple name

    System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 1:  sf1=20, sf
  }

  // Field declarations:
  static int sf1 = sf2 = 30;  // (7) Static field. Assignment to sf2 allowed
  static int sf2;              // (8) Static field
  int if1 = 5;                 // (9) Non-static field

  static {                     // (10) Static initializer block
    System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 2: sf1=30, sf

    int d = 2 * sf1;           // (11) OK. Read operation after declaration
    var e = sf1 = 50;          // (12) OK. Assignment to sf1 allowed

    System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 2:  sf1=50, sf
  }

  public static void main(String[] args) {
  }
}
```

Output from the program:

Click here to view code image

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

Example 10.4 gives an idea of the code rearrangement that the compiler does to facilitate the initialization for the class in Example 10.3. Field declarations from Example 10.3 are arranged first at (1) and (2) in Example 10.4, followed by a single `static` initializer block at (3). The `static` initializer block in Example 10.4 contains the code for the first `static` initializer block, followed by the initializer expression from the first `static` field declaration statement, and lastly the code from the second `static` initializer block in the order these initializers are declared in Example 10.3.

During class initialization, first the declarations of the `static` fields `sf1` and `sf2` at (1) and (2), respectively, result in them being created and initialized to their default value `0`. Not surprisingly, execution of the code in the `static` initializer block at (3) gives the same result as in Example 10.3.

Example 10.4 *Static Initializers and Order of Execution*

Click here to view code image

```
package refs2;

public class ForwardRefsSimulated {

  // Declaration of static fields:
  static int sf1;              // (1) Initialized to default value: 0
```

```
    static int sf2;              // (2) Initialized to default value: 0

    static {                     // (3) Static initializer block
      // Code from static block 1:

      System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
          sf1, sf2);             // Enter static block 1: sf1=0, sf2=0

      sf1 = 10;                  // (4) sf1 gets the value 10
      var b = sf1 = 20;          // (5) b and sf1 get the value 20
      int c = sf1;               // (6) c gets the value 20

      System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
          sf1, sf2);             // Exit static block 1:  sf1=20, sf2=0

      // Initializer expressions for field declaration:
      sf1 = sf2 = 30;            // (7) sf1 and sf2 get the value 30

      // Code from static block 2:
      System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
          sf1, sf2);             // Enter static block 2: sf1=30, sf2=30

      int d = 2 * sf1;           // (8) d gets the value 60
      var e = sf1 = 50;          // (9) e and sf1 get the value 50

      System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
          sf1, sf2);             // Exit static block 2:  sf1=50, sf2=30
    }

    public static void main(String[] args) {
    }
  }
```

Output from the program:

Click here to view code image

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

Exception Handling in Static Initializer Blocks

Exception handling in static initializer blocks is no different from that in static initializer expressions: Uncaught checked exceptions cannot be thrown. Code in initializers cannot throw *checked* exceptions. A `static` initializer block cannot be called directly. Therefore, any checked exceptions must be caught and handled in the body of the `static` initializer block; otherwise, the compiler will issue an error. Example 10.5 shows a `static` initializer block at (3) that catches and handles a checked exception in the `try-catch` block at (4).

Example 10.5 *Static Initializer Blocks and Exceptions*

Click here to view code image

```
// File: ExceptionInStaticInitBlocks.java
package version1;

class TooManyCellsException extends Exception {           // (1) Checked Except
  TooManyCellsException(String number) { super(number); }
}

//____________________________________________________________________
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells;                     // (2) No initializer expression

  static {                                        // (3) Static block
    try {                                         // (4) Handles checked exception
      if (noOfCells > 300)
        throw new TooManyCellsException(String.valueOf(noOfCells));
    } catch (TooManyCellsException e) {
      System.out.println("Exception handled: " + e);
      noOfCells = 300;
      System.out.println("No. of cells adjusted to " + noOfCells);
    }
    cells = new int[noOfCells];
  }
}
//____________________________________________________________________
public class ExceptionInStaticInitBlocks {
  public static void main(String[] args) {
    new Prison();
  }
}
```

Output from the program:

Click here to view code image

```
Exception handled: version1.TooManyCellsException: 365
No. of cells adjusted to 300
```

Static initializer blocks do not exactly aid code readability, and should be used sparingly, if at all. The code in the `static` initializer block at (3) in Example 10.5 can easily be refactored to instantiate the `static` array field `cells` at (2) using the `private static` method at (3) that handles the checked exception `TooManyCellsException`:

Click here to view code image

```
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells = initPrison();  // (2) Initializer expression

  //
  private static int[] initPrison() {         // (3) Private static method
    try {                                     // (4) Handles checked exception
```

```
        if (noOfCells > 300)
          throw new TooManyCellsException(String.valueOf(noOfCells));
      } catch (TooManyCellsException e) {
        System.out.println("Exception handled: " + e);
        noOfCells = 300;
        System.out.println("No. of cells adjusted to " + noOfCells);
      }
      return new int[noOfCells];
    }
  }
```

10.8 Instance Initializer Blocks

Just as static initializer blocks can be used to initialize `static` fields in a named class, Java provides the ability to initialize fields during object creation using instance initializer blocks. In this respect, such blocks serve the same purpose as constructors during object creation. The syntax of an instance initializer block is the same as that of a local block, as shown at (2) in the following code. The code in the local block is executed every time an instance of the class is created.

Click here to view code image

```
  class InstanceInitializers {
    long[] squares = new long[10];    // (1)
    // ...
    {                                 // (2) Instance Initializer
      for (int i = 0; i < squares.length; i++)
        squares[i] = i*i;
    }
    // ...
  }
```

The array `squares` of specified length is first created at (1); its creation is followed by the execution of the instance initializer block at (2) every time an instance of the class `InstanceInitializers` is created. Note that the instance initializer block is not contained in any method. A class can have more than one instance initializer block, and these (and any instance initializer expressions in instance field declarations) are executed in the order they are specified in the class.

Declaration Order of Instance Initializers

Analogous to the other initializers discussed earlier, an instance initializer block cannot make a forward reference to a field by its simple name in a read operation as that would violate the declaration-before-reading rule. However, using the `this` keyword to access a field is not a problem.

The class below has an instance initializer block at (1) with forward references to the fields `i`, `j`, and `k` that are declared at (7), (8), and (9), respectively. These fields are accessed using the `this` reference in *read* operations at (3), (4), (5), and (6). Using the simple name of these fields at (3), (4), (5), and (6) to access their values will violate the declare-before-use rule, resulting in compile-time errors—regardless of whether the fields are declared with initializer expressions or not, or whether they are final or not. The fields `i` and `j` are accessed at (2) in *write* operations, which are permitted using the simple name. However, care must be exercised to ensure that the fields are initialized correctly. At (3), (4), and (5), the fields `i` and `j` have the value `10`. However, when the initializer expressions are evaluated in the instance field declarations, the value of `j` will be set to `100`.

Click here to view code image

```
public class InstanceInitializersII {

  { //Instance initializer with forward references.   (1)
    i = j = 10;                                    // (2) Permitted.
    int result = this.i * this.j;                  // (3) i is 10, j is 10.
    System.out.println(this.i);                    // (4) 10
    System.out.println(this.j);                    // (5) 10
    System.out.println(this.k);                    // (6) 50
  }
  // Instance field declarations.
  int i;                 // (7) Field declaration without initializer expressi
  int j = 100;           // (8) Field declaration with initializer expression.
  final int k = 50;      // (9) Final instance field with constant expression.

}
```

Example 10.6 illustrates some additional subtle points regarding instance initializer blocks. In Example 10.6, an illegal forward reference occurs in the code at (4), which attempts to read the value of the field `nsf1` before it is declared. The read operation at (11) occurs after the declaration, and therefore, is allowed. Forward reference made on the left-hand side of the assignment is always allowed, as shown at (2), (3), (5), and (7).

Declaring local variables using the reserved word `var` in instance initializer blocks is shown at (5) and (12) in Example 10.6.

Example 10.6 *Instance Initializers and Forward References*

Click here to view code image

```
public class NonStaticForwardReferences {

  {                        // (1) Instance initializer block.
    nsf1 = 10;             // (2) OK. Assignment to nsf1 allowed.
    nsf1 = sf1;            // (3) OK. Static field access in non-static contex
    //  int a = 2 * nsf1;  // (4) Not OK. Read operation before declaration.
    var b = nsf1 = 20;     // (5) OK. Assignment to nsf1 allowed.
    int c = this.nsf1;     // (6) OK. Not accessed by simple name.
  }

  int nsf1 = nsf2 = 30;    // (7) Non-static field. Assignment to nsf2 allowed
  int nsf2;                // (8) Non-static field.
  static int sf1 = 5;      // (9) Static field.

  {                        // (10) Instance initializer block.
    int d = 2 * nsf1;      // (11) OK. Read operation after declaration.
    var e = nsf1 = 50;     // (12) OK. Assignment to nsf1 allowed.
  }

  public static void main(String[] args) {
    NonStaticForwardReferences objRef = new NonStaticForwardReferences();
    System.out.println("nsf1: " + objRef.nsf1);
    System.out.println("nsf2: " + objRef.nsf2);
  }
}
```

Output from the program:

```
nsf1: 50
nsf2: 30
```

As in an instance initializer expression, the keywords `this` and `super` can be used to refer to the current object in an instance initializer block.

As in a static initializer block, the `return` statement is also not allowed in instance initializer blocks.

An instance initializer block can be used to factor out common initialization code that will be executed regardless of which constructor is invoked. A typical usage of an instance initializer block is in anonymous classes (§9.6, p.521), which cannot declare constructors, but can instead use instance initializer blocks to initialize fields. In Example 10.7, the anonymous class defined at (1) uses an instance initializer block at (2) to initialize its fields.

Example 10.7 *Instance Initializer Block in Anonymous Class*

Click here to view code image

```
// File: InstanceInitBlock.java
class Base {
  protected int a;
  protected int b;
  void print() { System.out.println("a: " + a); }
}
//____________________________________________________________________________
class AnonymousClassMaker {
  Base createAnonymous() {
    return new Base() {           // (1) Anonymous class
      {                           // (2) Instance initializer
        a = 5; b = 10;
      }

      @Override
      void print() {
        super.print();
        System.out.println("b: " + b);
      }
    };  // end anonymous class
  }
}
//____________________________________________________________________________
public class InstanceInitBlock {
  public static void main(String[] args) {
    new AnonymousClassMaker().createAnonymous().print();
  }
}
```

Output from the program:

```
a: 5
b: 10
```

Exception Handling and Instance Initializer Blocks

Exception handling in instance initializer blocks is similar to that in static initializer blocks. Example 10.8 shows an instance initializer block at (3) that catches and handles a checked exception in the `try-catch` block at (4). Another instance initializer block at (5) throws an unchecked exception at (6). The runtime system handles the runtime exception, printing the stack trace and terminating the program.

Exception handling in instance initializer blocks differs from that in static initializer blocks in the following aspect: The execution of an instance initializer block can result in an uncaught checked exception, provided the exception is declared in the `throws` clause of *every* constructor in the class. Static initializer blocks cannot allow this, since no constructors are involved in class initialization. Instance initializer blocks in anonymous classes have even greater freedom: They can throw any exception.

Example 10.8 *Exception Handling in Instance Initializer Blocks*

Click here to view code image

```
// File: ExceptionsInInstBlocks.java
class RoomOccupancyTooHighException
      extends Exception {}                     // (1) Checked exception
class BankruptcyException
      extends RuntimeException {}              // (2) Unchecked exception
//_______________________________________________________________________
class Hotel {
  // Instance Members
  private boolean bankrupt         = true;
  private int     noOfRooms        = 215;
  private int     occupancyPerRoom = 5;
  private int     maxNoOfGuests;

  {                                            // (3) Instance initializer bloc
    try {                                      // (4) Handles checked exception
      if (occupancyPerRoom > 4)
        throw new RoomOccupancyTooHighException();
    } catch (RoomOccupancyTooHighException exception) {
      System.out.println("ROOM OCCUPANCY TOO HIGH: " + occupancyPerRoom);
      occupancyPerRoom = 4;
    }
    maxNoOfGuests = noOfRooms * occupancyPerRoom;
  }

  {                                            // (5) Instance initializer bloc
    if (bankrupt)
      throw new BankruptcyException();          // (6) Throws unchecked excepti
  }   // ...
}
//_______________________________________________________________________
public class ExceptionsInInstBlocks {
  public static void main(String[] args) {
    new Hotel();
  }
}
```

Output from the program:

Click here to view code image

```
ROOM OCCUPANCY TOO HIGH: 5
Exception in thread "main" BankruptcyException
        at Hotel.<init>(ExceptionsInInstBlocks.java:27)
        at ExceptionsInInstBlocks.main(ExceptionsInInstBlocks.java:33)
```

10.9 Constructing Initial Object State

Object initialization involves constructing the initial state of an object when it is created by the `new` operator. First the fields are initialized to their default values (§3.4, p.103)—whether they are subsequently given non-default initial values or not—and then the constructor is invoked. This can lead to *local* chaining of constructors. The invocation of the constructor at the end of the local chain of constructor invocations results in the following actions, before the constructor’s execution resumes:

- Implicit or explicit invocation of the superclass constructor. Constructor chaining ensures that the state from the object’s superclasses is constructed first (§5.3, p.209).
- Initialization of the instance fields by executing their instance initializer expressions and any instance initializer blocks, in the order they are specified in the class declaration.

Example 10.9 illustrates object initialization. The `new` operator is used at (8) to create an object of `SubclassB`. The no-argument constructor `SubclassB()` at (2) uses the `this()` construct to locally chain to the non-zero argument constructor at (3). This constructor then leads to an implicit call of the superclass constructor. As can be seen from the program output, the execution of the superclass’s constructor at (1) reaches completion first. This is followed by the execution of the instance initializer block at (4) and the instance initializer expression at (6). Then the execution of the body of the non-zero argument constructor at (3) resumes. Finally, the no-argument constructor completes its execution, thereby completing the construction of the object state.

Note that the instance initializers are executed in the order they are specified in the class declaration. The forward reference to the field `value` at (5) is legal because the usage of the field `value` is on the left-hand side of the assignment (it does not violate the declaration-before-reading rule). The default value of the field `value` is overwritten by the instance initializer block at (5). The field `value` is again overwritten by the instance initializer expression at (6), and finally by the non-zero argument constructor at (3).

Example 10.9 *Object State Construction*

Click here to view code image

```
// File: ObjectConstruction.java
class SuperclassA {
  public SuperclassA() {                     // (1) Superclass constructor
    System.out.println("Constructor in SuperclassA");
  }
}
//______________________________________________________________________________
class SubclassB extends SuperclassA {

  SubclassB() {                              // (2) No-argument constructor
    this(3);
    System.out.println("No-argument constructor in SubclassB");
  }

  SubclassB(int i) {                         // (3) Non-zero argument construc
    System.out.println("Non-zero argument constructor in SubclassB");
    value = i;
  }

  {                                          // (4) Instance initializer block
    System.out.println("Instance initializer block in SubclassB");
    value = 2;                               // (5)
  }

  int value = initializerExpression();       // (6) Instance field declaration

  private int initializerExpression() {      // (7)
    System.out.println("Instance initializer expression in SubclassB");
    return 1;
  }
}
//____________________________________________________________________________
public class ObjectConstruction {
  public static void main(String[] args) {
    SubclassB objRef = new SubclassB();         // (8)
    System.out.println("value: " + objRef.value);
  }
}
```

Output from the program:

Click here to view code image

```
Constructor in SuperclassA
Instance initializer block in SubclassB
Instance initializer expression in SubclassB
Non-zero argument constructor in SubclassB
No-argument constructor in SubclassB
value: 3
```

Some care should be exercised when writing constructors for non-`final` classes, since the object that is constructed might be a subclass instance. Example 10.10 shows a situation where use of overridden methods in *superclass* initializers and constructors can give unexpected results. The example intentionally uses the `this` reference to underline that the instance methods and constructors are invoked on the current object, and that the constructor call results in the initialization of the object state, as expected.

The program output from Example 10.10 shows that the field `superValue` at (1) in `SuperclassA` never gets initialized explicitly when an object of `SubclassB` is created at (8). The `SuperclassA` constructor at (2) does have a call to a method that has the name `doValue` at (3). A method with such a name is defined in `SuperclassA` at (4), but is also overridden in `SubclassB` at (7). The program output indicates that the method `doValue()` from `SubclassB` is called at (3) in the `SuperclassA` constructor. The implementation of the method `doValue()` at (4) never gets executed when an object of `SubclassB` is created. Method invocation always determines the implementation of the method to be executed, based on the *actual* type of the object. Keeping in mind that it is an object of `SubclassB` that is being initialized, the call to the method named `doValue` at (3) results in the method from `SubclassB` being executed. This can lead to unintended results. The overriding method `doValue()` at (7) in `SubclassB` can access the field `value` declared at (5) before its initializer expression has been executed; thus the method invoked can access the state of the object *before* this has been completely initialized. The value `0` is then printed, as the field `value` has not yet been initialized with the value `800` when the superclass constructor is executed.

Class initialization takes place before any instance of the class can be created or a static method of the class can be invoked. A superclass is initialized before its subclasses are initialized. Initializing a class involves initialization of the static fields by executing their static initializer expressions and any static initializer blocks.

Initialization of an interface involves execution of any static initializer expressions for the `public static final` fields declared in the interface. An interface cannot specify instance initializer expressions because it has no instance fields, nor can it specify any initializer blocks because it cannot be instantiated.

Example 10.10 *Initialization Anomaly under Object State Construction*

Click here to view code image

```
// File: ObjectInitialization.java
class SuperclassA {
  protected int superValue;                              // (1)
  SuperclassA() {                                        // (2)
    System.out.println("Constructor in SuperclassA");
    this.doValue();                                      // (3)
  }
  void doValue() {                                       // (4)
    this.superValue = 911;
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
}
//____________________________________________________________________________
class SubclassB extends SuperclassA {
  private int value = 800;                               // (5)
  SubclassB() {                                          // (6)
    System.out.println("Constructor in SubclassB");
    this.doValue();
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
  @Override
  void doValue() {                                       // (7)
    System.out.println("value (from SubclassB): " + this.value);
  }
}
//______________________________________________________________________________
public class ObjectInitialization {
  public static void main(String[] args) {

    System.out.println("Creating an object of SubclassB.");
    new SubclassB();                                     // (8)
  }
}
```

Output from the program:

Click here to view code image

```
Creating an object of SubclassB.
Constructor in SuperclassA
value (from SubclassB): 0
Constructor in SubclassB
value (from SubclassB): 800
superValue (from SuperclassA): 0
```

Review Questions

10.6 Given the following class, which of these static initializer blocks can be independently inserted at (1)?

Click here to view code image

```
public class MyClass {
  private static int count = 5;
  static final int STEP = 10;
  boolean alive;

  // (1) INSERT STATIC INITIALIZER BLOCK HERE
}
```

Select the three correct answers.

a. `static { alive = true; count = 0; }`

b. `static { STEP = count; }`

c. `static { count += STEP; }`

d. `static ;`

e. `static { }`

f. `static { count = 1; }`

10.7 What will be the result of compiling and running the following program?

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    MyClass obj = new MyClass(n);
  }


  static int i = 5;
  static int n;
  int j = 7;
  int k;

  public MyClass(int m) {
    System.out.println(i + ", " + j + ", " + k + ", " + n + ", " + m);
  }
  { j = 70; n = 20; } // Instance initializer block
  static { i = 50; }  // Static initializer block
}
```

Select the one correct answer.

- **a.** The code will fail to compile because of the instance initializer block.
- **b.** The code will fail to compile because of the static initializer block.
- **c.** The code will compile and print `50, 70, 0, 20, 0` at runtime.
- **d.** The code will compile and print `50, 70, 0, 20, 20` at runtime.
- **e.** The code will compile and print `5, 70, 0, 20, 0` at runtime.
- **f.** The code will compile and print `5, 70, 0, 20, 20` at runtime.
- **g.** The code will compile and print `5, 7, 0, 20, 0` at runtime.
- **h.** The code will compile and print `5, 7, 0, 20, 20` at runtime.

10.8 Given the following class, which instance initializer block inserted independently at (1) will allow the class to be compiled?

Click here to view code image

```
public class FirstClass {
  static int gap = 10;
  double length;
  final boolean active;

  // (1) INSERT CODE HERE
}
```

Select the one correct answer.

- **a.** `instance { active = true; }`

b. `FirstClass { gap += 5; }`

c. `{ gap = 5; length = (active ? 100 : 200) + gap; }`

d. `{ }`

e. `{ length = 4.2; }`

f. `{ active = (gap > 5); length = 5.5 + gap;}`

10.9 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Initialization {
  private static String msg(String msg) {
    System.out.println(msg);
    return msg;
  }

  public Initialization() { m = msg("1"); }

  { m = msg("2"); }

  String m = msg("3");

  public static void main(String[] args) {
    Object obj = new Initialization();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile and print `1`, `2`, and `3` at runtime.

c. The program will compile and print `2`, `3`, and `1` at runtime.

d. The program will compile and print `3`, `1`, and `2` at runtime.

e. The program will compile and print `1`, `3`, and `2` at runtime.

10.10 What is the result of executing the following program?

Click here to view code image

```
public class Music {
  static {
    System.out.print("-C-");
  }
  {
    System.out.print("-D-");
  }
  public Music(){
    System.out.print("-E-");
  }
}
```

Click here to view code image

```
public class Song extends Music {
  static {
    System.out.print("-F-");
  }
  {
    System.out.print("-G-");
  }
  public Song(){
      System.out.print("-A-");
    }
  }
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Music x1 = new Song();
    Song x2 = new Song();
  }
}
```

Select the one correct answer.

- **a.** `-C--D--E--F--G--A--G--A--G--A-`
- **b.** `-C--D--E--F--G--A--D--E--G--A-`
- **c.** `-C--F--D--E--G--A--D--E--G--A-`
- **d.** `-C--F--D--G--E--A--D--G--E--A-`
- **e.** `-C--F--E--D--A--G--E--D--A--G-`

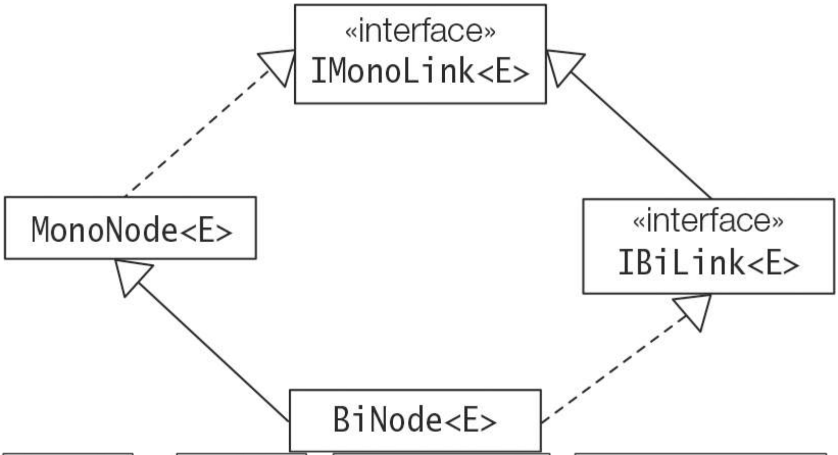
10.11 Which labeled lines in the following code can be independently *uncommented* by removing the `//` characters at the beginning of a line, such that the code will still compile?

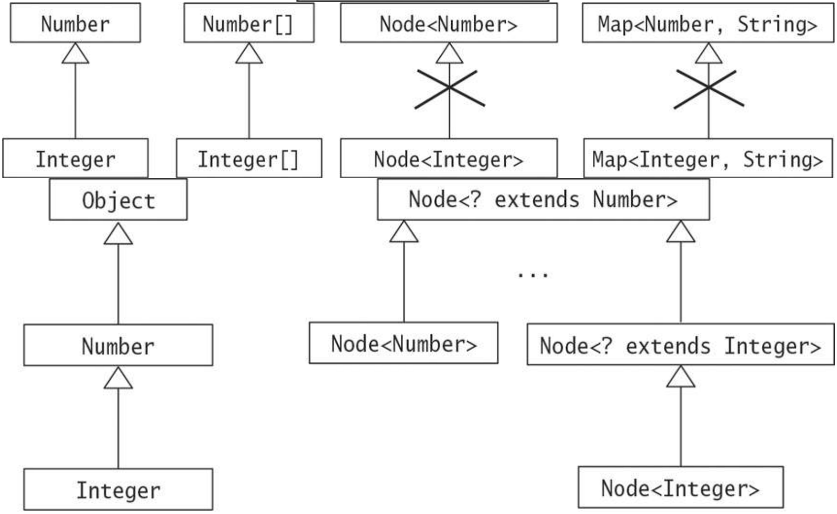
Click here to view code image

```
class GeomInit {
//int width = 14;             // (1)
  {
//  area = width * height;    // (2)
  }

  int width = 37;
  {
//  height = 11;              // (3)
  }

  int height, area;
//area = width * height;      // (4)
  {
//  int width = 15;           // (5)
    area = 100;

  }
}
```

Select the two correct answers.

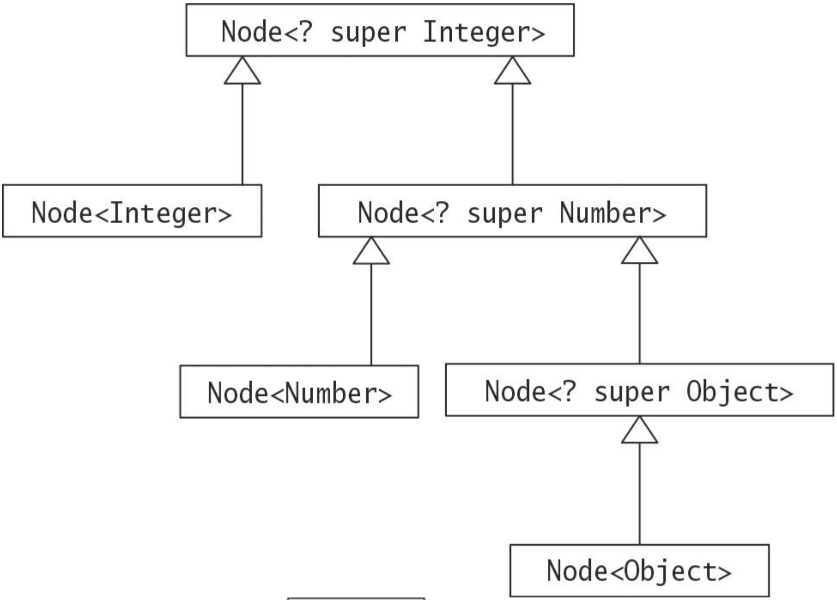
- **a.** Line (1)
- **b.** Line (2)
- **c.** Line (3)
- **d.** Line (4)
- **e.** Line (5)

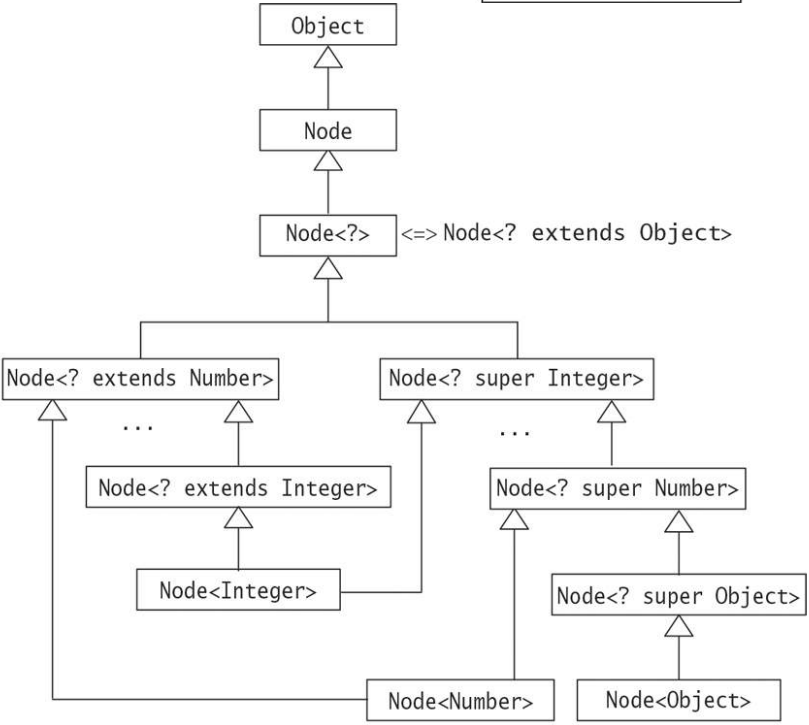

«interface»
IMonoLink<E>
MonoNode<E>
«interface»
IBiLink<E>
BiNode<E>


Number
Number[]
Integer
Integer[]
Object
Number
Integer
Node<Number>
Map<Number, String>
Node<Integer>
Map<Integer, String>
Node<? extends Number>
...
Node<Number>
Node<? extends Integer>
Node<Integer>

(a) Inheritance hierarchy

(b) Partial type hierarchy


Node<? super Integer>
Node<Integer>
Node<? super Number>
Node<Number>
Node<? super Object>
Node<Object>


Object
Node
Node<?>
<=> Node<? extends Object>
Node<? extends Number>
...
Node<? super Integer>
...
Node<? extends Integer>
Node<? super Number>
Node<Integer>
Node<? super Object>
Node<Number>
Node<Object>

All widening reference conversions are type-safe.

All narrowing reference conversions require an explicit cast, except for the following:

- *Narrowing reference conversion from* `Node` *to* `Node<?>`*, or from* `Node<?>` *to* `Node<? extends Object>`*, is safe.*
- *Narrowing reference conversion from* `Node` *to* `Node<? extends Object>`*, or to any subtype below* `Node<?>`*, results in an unchecked conversion warning.*

Generics 11

Chapter Topics

- Understanding the relationship between generic types, parameterized types, and raw types
- Declaring generic types (normal classes, record classes, enum types and interfaces) and parameterized types
- Use of the diamond operator (`<>`) when creating objects of generic classes
- Using the restricted keyword `var` to declare local generic type references
- Extending generic types
- Understanding the folly of mixing generic code and legacy code
- Understanding the significance of unchecked warnings on type-safety
- Understanding subtype relationships for wildcards
- Understanding type hierarchy for wildcard parameterized types
- Understanding widening and narrowing reference conversions in type hierarchy of wildcard parameterized types
- Understanding restrictions on set and get operations when using references of wildcard parameterized types
- Using bounded type parameters
- Understanding how to implement a generic class that is

also `Iterable`

- Understanding wildcard capture
- Programming with wildcard parameterized types
- Writing generic methods and constructors
- Understanding the implications of type erasure
- Understanding how overloading and overriding work with generics
- Understanding reifiable and non-reifiable types, and their role in generics
- Understanding the limitations and restrictions that generics place on instance tests, casting, arrays, varargs, and exception handling

Java SE 17 Developer Suppler	
[12.3] Use generics, including wildcards.	§11.1, p. 565 to §11.4, p. 579
Note that this is a supplementary objective in the description of the Java SE 17 Developer exam.	
Java SE 11 Developer Exam O	
[5.1] Use generics, including wildcards	§11.1, p. 565 to §11.13, p. 623

Introduction of generics in Java was a major step in the evolution of the language. Extending the language with generics provides even stronger type checking at compile time and allows implementing generic algorithms, especially for managing *containers* of data, like the collections discussed in Chapter 15.

Generics in Java is a large topic with some subtle details. Extending the language with generics has consequences for many aspects of the language. Familiarity with topics covered in this chapter will be beneficial in both using and implementing generic types.

The basics of generics are covered in §11.1, §11.2, and §11.3, which introduce the terminology, the underlying concepts, and the motivation for using generics for implementing collections.

On a first reading, §11.4 to §11.8 should be read to become familiar with wildcards and the flexibility they provide in defining parameterized types, and with generic methods and constructors. The reader should come back to these topics for an in-depth study when the need arises.

For an even deeper dive into generics, we suggest studying the topics from §11.9 onward: wildcard capture and wildcard parameterized types, implications of type erasure, and restrictions on generic types.

11.1 Introducing Generics

Generics allow classes and interfaces, as well as methods and constructors, to be *parameterized* with *type information*. An *abstract data type* (ADT) defines both the *types* of objects and the *operations* that can be performed on these objects. Generics allow us to specify the types used by the ADT so that the *same* definition of an ADT can be used on *different* types of objects.

Generics in Java are a way of providing type information in ADTs so that the compiler can guarantee type-safety of operations at runtime. Generics are implemented as compile-time transformations, with negligible impact on the JVM. The generic type declaration is compiled once into a single Java class file, and the use of the generic type is checked against this file. Also, no extraneous Java class files are generated for each use of the generic type.

The primary benefits of generics are increased language expressiveness with improved type-safety, resulting in improved robustness and reliability of code. Generics avoid verbosity of using casts in many contexts, thus improving code clarity. Since the compiler guarantees type-safety, this eliminates the necessity of explicit type checking and casting at runtime.

One major goal when introducing generics in Java has been backward compatibility with legacy code (i.e., non-generic code). Interoperability with legacy code and the lack of generic type information at runtime largely determine how generics work in Java. Many of the restrictions on generics in Java can be attributed to these two factors.

Generics are used extensively in implementing the Java Collections Framework. An overview of Chapter 15 on collections and maps is therefore recommended as many of the examples in this chapter make use of generic types from this framework.

Before the introduction of generics in Java, a general implementation of a collection maintained its objects by using references of the type `Object`. The bookkeeping of the actual type of the objects fell on the client code. Example 11.1 illustrates this approach. It implements a *self-referential data structure* called a *node*.

Each node holds a data value and a reference to another node. Such data structures form the basis for building *linked data structures*.

Example 11.1 *A Legacy Class*

Click here to view code image

```
class LegacyNode {
  private Object     data;    // The value in the node
  private LegacyNode next;    // The reference to the next node.
  LegacyNode(Object data, LegacyNode next) {
    this.data = data;
    this.next = next;
  }
  public void       setData(Object obj)      { this.data = obj; }
  public Object     getData()                { return this.data; }
  public void       setNext(LegacyNode next) { this.next = next; }
  public LegacyNode getNext()                { return this.next; }
  @Override public String toString() {
    return this.data + (this.next == null? "" : ", " + this.next);
  }
}
```

The class `LegacyNode` can be used to create a linked list with arbitrary objects:

Click here to view code image

```
LegacyNode node1 = new LegacyNode(4, null);       // 4 --> null
LegacyNode node2 = new LegacyNode("July", node1); // "July" --> 4 --> null
```

Primitive values are encapsulated in corresponding wrapper objects. If we want to retrieve the data from a node, the data is returned via an `Object` reference:

Click here to view code image

```
Object obj = node2.getData();
```

In order to access type-specific properties or behavior of the fetched object, the reference value in the `Object` reference must be converted to the right type. To avoid a `ClassCastException` at runtime when applying the cast, we must make sure that the object referred to by the `Object` reference is of the right type. All that can be accomplished by using the `instanceof` pattern match operator:

Click here to view code image

```
if (obj instanceof String str) {
  System.out.println(str.toUpperCase()); // Method specified in the String cla
}
```

The approach outlined above places certain demands on how to use the class `LegacyNode` to create and maintain linked structures. For example, it is the responsibility of the client code to ensure that the objects being put in nodes are of the same type. Implementing classes for specific types of objects is not a good solution. First, it can result in code duplication, and second, it is not always known in advance what types of objects will be put in the nodes. Generic types offer a better solution, where one generic class is defined and specific reference types are supplied each time we want to instantiate the class.

11.2 Generic Types and Parameterized Types

We first introduce the basic terminology and concepts relating to generics in Java. Note that the discussion here on generic and parameterized types also applies to enum types (§5.13, p. 287) and record classes (§5.14, p. 299).

Generic Types

A *generic type* is a reference type that defines a *list of formal type parameters* or *type variables* that must be provided before it can be used as a type. Example 11.2 declares a generic type which, in this case, is a *generic class* called `Node<E>` that allows nodes of specific types to be maintained. It has only one formal type parameter, `E`, that represents the type of the data in a node.

```
class Node<E> {
...
}
```

The formal type parameter `E` does not explicitly specify a type, but serves as a placeholder for a type to be defined in an invocation of the generic type. The formal type parameters of a generic type are specified within angle brackets, < >, immediately after the class name. A type parameter is an unqualified identifier. If a generic class has several formal type parameters, these are specified as a comma-separated list, `<T1, T2, …, Tn>`. It is quite common to use one-letter names for formal type parameters; a convention that we will follow in this book. For example, `E` is used for the type of elements in a collection, `K` and `V` are used for the type of the keys and the type of the values in a map, and `T` is used to represent an arbitrary type.

As a starting point for declaring a generic class, we can begin with a class where the `Object` type is utilized to generalize the use of the class. In Example 11.2, the declaration of the generic class `Node<E>` uses `E` in all the places where the type `Object` was used in the declaration of the class `LegacyNode` in Example 11.1. From the declaration of the class `Node<E>`, we can see that the formal type `E` is used like a reference type in the class body: as a field type at (1), as a return type at (5), and as a parameter type in the methods at (4) to (8). Use of the class name in the generic class declaration is parameterized by the type parameter ((2), (6), (7)), with one notable exception: The formal type parameter is *not* specified after the class name in the constructor declaration at (3). Which actual reference type the formal type parameter `E` represents is not known in the generic class `Node<E>`. Therefore, we can only call methods that are inherited from the `Object` class on the field `data`, as these methods are inherited by all objects, regardless of their object type. One such example is the call to the `toString()` method in the method declaration at (8).

The scope of the type parameter `E` of the generic type includes any non-static inner classes, but excludes any static member types—the parameter `E` cannot be accessed in static context. It also excludes any nested generic declarations where the same name is redeclared as a formal type parameter. Shadowing of type parameter names should be avoided.

Example 11.2 *A Generic Class for Nodes*

Click here to view code image

```
class Node<E> {
  private E              data;    // Data                         (1)
  private Node<E>        next;    // Reference to next node       (2)
  Node(E data, Node<E> next) {                                 // (3)
    this.data = data;
    this.next = next;
  }
  public void    setData(E data)        { this.data = data; }   // (4)
  public E       getData()              { return this.data; }   // (5)
  public void    setNext(Node<E> next) { this.next = next; }   // (6)
  public Node<E> getNext()              { return this.next; }   // (7)
  @Override public String toString() {                         // (8)
    return this.data.toString() +
           (this.next == null ? "" : ", " + this.next.toString());
  }
}
```

Some Restrictions on the Use of Type Parameters in a Generic Type

A constructor declaration in a generic class cannot specify the formal type parameters of the generic class in its constructor header after the class name:

Click here to view code image

```
class Node<E> {
  ...
  Node<E>() { ... }                          // Compile-time error!
  ...
}
```

A formal type parameter cannot be used to create a new instance, as it is not known which concrete type it represents. The following code in the declaration of the `Node<E>` class would be illegal:

Click here to view code image

```
E ref = new E();                             // Compile-time error!
```

A formal type parameter is a *non-static type*. It cannot be used in a static context, for much the same reason as an instance variable cannot be used in a static context: It is associated with objects. The compiler will report errors at (1), (2), and (3) in the code below:

Click here to view code image

```
class Node<E> {
  private static E e1;                           // (1) Compile-time error!
  public  static E oneStaticMethod(E e2) {       // (2) Compile-time error!
    E e3;                                        // (3) Compile-time error!
    System.out.println(e3);
  }
  // ...
}
```

Parameterized Types

A *parameterized type* (also called a *type instance*) is an *invocation* or *instantiation* of a generic type that is a specific usage of the generic type where the formal type parameters are replaced by *actual type parameters*. Analogy with method declarations and method invocations can be helpful in understanding the relationship between generic types and parameterized types. We pass actual parameters in a method invocation to execute a method. In the case of a generic type invocation, we pass actual *type* parameters in order to instantiate a generic type.

We can declare references and create objects of parameterized types, and call methods on these objects, in much the same way as we use non-generic classes.

Click here to view code image

```
Node<Integer> intNode = new Node<Integer>(2020, null);
```

The actual type parameter `Integer`, explicitly specified in the declaration statement above, binds to the formal type parameter `E` in Example 11.2. The compiler treats the parameterized type `Node<Integer>` as a *new type*. The parameterized type `Node<Integer>` constrains the generic type `Node<E>` to `Integer` objects, thus implementing homogenous nodes with `Integer`s. The reference `intNode` can only refer to a `Node` of `Integer`. The node created can only be used to store an object of this concrete type.

Methods can be called on objects of parameterized types:

Click here to view code image

```
Integer iRef = intNode.getData();            // Integer object with int value
```

In the method call above, the actual type parameter is determined from the type of the reference used to make the call. The type of the `intNode` reference is `Node<Integer>`; therefore, the actual type parameter is `Integer`. The method header is `Integer getData()`, meaning that the method will return a value of type `Integer`. The compiler checks that the return value can be assigned. As the compiler guarantees that the return value will be an `Integer` and can be assigned, no explicit cast or runtime check is necessary. The compiler actually inserts the necessary cast. Here are some more examples of calling methods of parameterized types:

Click here to view code image

```
intNode.setData(2020);                            // Ok.
intNode.setData("TwentyTwenty");                  // (1) Compile-time error!
intNode.setNext(new Node<Integer>(2019, null));   // (2020, (2019, null))
intNode.setNext(new Node<String>("Hi", null));    // (2) Compile-time error!
```

In the method calls shown above, the compiler determines that the actual type parameter is `Integer`. The method signatures are `setData(Integer)` and `setNext(Node<Integer>)`. As expected, we get a compile-time error when we try to pass an argument that is not compatible with the parameter type in the method declarations; for example, at (1) and (2). The parameterized types `Node<Integer>` and `Node<String>` are two *unrelated* types. The compiler reports any inconsistent use of a parameterized type so that errors can be caught earlier at compile time and the use of explicit casts in the source code is minimized, as evident from (3) and (4), respectively.

Click here to view code image

```
Node<String> strNode = new Node<String>("Hi", null);
intNode = strNode;                         // (3) Compile-time error!
String str = strNode.getData();            // (4) No explicit cast necessary.
```

The Diamond Operator (<>)

In the object creation expression of the `new` operator, the actual type parameter was explicitly specified after the class name—in contrast to the constructor declaration.

Click here to view code image

```
Node<String> lst = new Node<String>("Hi", null); // Explicit actual type param
```

The actual type parameters can be omitted, but not the angle brackets (<>), if the compiler can infer the actual type parameters from the context of the object creation expression. The angle brackets with no actual parameters (<>) are commonly referred to as the *diamond operator*.

Click here to view code image

```
Node<String> lst = new Node<>("Hi", null); // Actual type parameter inferred.
```

In the object creation expression above, the compiler performs automatic type inference to infer that the actual type parameter in the expression must be `String`. The compiler is able to infer the actual type parameter from the type information of the constructor call arguments.

Click here to view code image

```
new Node<>(null, null);  // Actual type parameter: Object.
```

In the code below, the compiler uses the type information of the variable on the left-hand side to infer the actual type parameter of the object creation expression, thereby ensuring compatibility with the *target type* on the left-hand side of the assignment.

Click here to view code image

```
Node<String>  strNode = new Node<>(null, null); // Actual type parameter: Stri
Node<Integer> intNode = new Node<>(null, null); // Actual type parameter: Inte
Node<Number>  numNode = new Node<>(null, null); // Actual type parameter: Numb
Node<Number>  lstNode = new Node<>(2021, null); // Actual type parameter: Numb
```

In the last declaration, the `int` value `2021` is boxed into an `Integer` object that can be assigned to a reference of its superclass `Number`. In other words, the signature of the constructor call is `Node<Number>(Number, Node<Number>)`.

Given the following scenario:

Click here to view code image

```
// (1) Method declaration with parameterized type as formal parameter.
void find(Node<String> node) { /* ... */ }
...
// (2) Method call where actual argument uses diamond operator.
find(new Node<>(null,null));  // Actual type parameter: Object or String?
```

The compiler takes the target type (`Node<String>`) in the method declaration into consideration, correctly inferring the actual type parameter to be `String`, in order for the actual and the formal parameters in the call to be assignment compatible.

A single diamond operator must replace the entire actual type parameter list in the object creation expression. In the first declaration below, the compiler infers that the actual type parameter list is `<String, List<Integer>>`.

Click here to view code image

```
HashMap<String, List<Integer>> map = new HashMap<>();
HashMap<String, List<Integer>> map = new HashMap<String,<>>(); // Error!
```

If the actual type parameters are not specified and the diamond operator is omitted, the compiler issues an *unchecked conversion warning*—that is, the code will compile, but all bets are off at runtime. Below, the reference type `Node` in the object creation expression is interpreted as a *raw type*. Implications of interoperability between generic types and raw types are discussed on p. 575.

Click here to view code image

```
Node<String> rawNode = new Node("Hi", null); // Unchecked conversion warning!
```

The diamond operator can be used to instantiate an anonymous class (see p. 633).

Click here to view code image

```
new Node<>("Hi", null) {/* ... */}; // Parameterized type: Node<String>
new Node("Hi", null)   {/* ... */}; // Raw type: Node
```

Parameterized Local Variable Type Inference

Consider the four local variable declarations shown below. The first three declarations are equivalent, as they create a `Node<String>` object whose `data` and `next` fields are `null`. In declaration (1), since the type `String` is explicitly specified in the object creation expression, the actual type parameter is deduced to be `String`. In declaration (2), the diamond operator is specified in the object creation expression, but the compiler is able to infer that the actual type parameter is `String` from the left-hand side of the declaration —that is, the compiler considers the context of the whole declaration. In the case of the parameterized local variable declarations with `var`, both the actual type parameters and the parameterized type denoted by `var` are inferred from the object creation expression.

Click here to view code image

```
// Parameterized local variable declarations:
Node<String> node1 = new Node<String>(null, null); // (1) Node of String
Node<String> node2 = new Node<>(null, null);       // (2) Node of String
var node3 = new Node<String>(null, null);          // (3) Node of String
var node4 = new Node<>(null, null);                // (4) Node of Object
```

In declaration (3), as `String` is explicitly specified in the object creation expression, the actual type parameter is inferred to be `String` and the parameterized type of `node3` denoted by `var` is inferred to be `Node<String>`.

However, in declaration (4), the diamond operator is used in the object creation expression. In this case, the actual type parameter is inferred to be `Object` and the parameterized type of `node4` denoted by `var` is inferred to be `Node<Object>`. Adequate type information should be provided in the object creation expression when declaring parameterized local variables with `var` in order to avoid unexpected types being inferred for the actual parameter types and the parameterized type denoted by `var`.

Generic Interfaces

Generic types also include generic interfaces, which are declared analogous to generic classes. The specification of formal type parameters in a generic interface is the same as in a generic class. Example 11.3 declares a generic interface that defines the reference type `IMonoLink<E>` for objects that store a data value of type `E`.

Example 11.3 *A Generic Interface and Its Implementation*

Click here to view code image

```
interface IMonoLink<E> {
  void         setData(E data);
  E            getData();
  void         setNext(IMonoLink<E> next);
  IMonoLink<E> getNext();
}

class MonoNode<E> implements IMonoLink<E> {
  private E            data;    // Data
  private IMonoLink<E> next;    // Reference to next node                    (1

  MonoNode(E data, IMonoLink<E> next) {                                   // (2
    this.data = data;
    this.next = next;
  }

  @Override public void setData(E data) { this.data = data; }
  @Override public E    getData()       { return this.data; }
  @Override public void setNext(IMonoLink<E> next) { this.next = next; } // (3
  @Override public IMonoLink<E> getNext()          { return this.next; } // (4
  @Override public String toString() {
    return this.data.toString() + (this.next == null? "" : ", " + this.next);
  }
}
```

A generic interface can be implemented by a generic (or a non-generic) class:

Click here to view code image

```
class MonoNode<E> implements IMonoLink<E> {
  // ...
}
```

Note that the construct `<E>` is used in two different ways in the class header. The first occurrence of `<E>` *declares* `E` to be a type parameter, and the second occurrence of `<E>` *parameterizes* the generic interface with this type parameter. The declare-before-use rule also applies to type parameters. The version of the `MonoNode` class in Example 11.3 differs from the `Node` class in Example 11.2 at (1), (2), (3), and (4). These changes were necessary to make the `MonoNode<E>` class compliant with the `IMonoLink<E>` interface.

A generic interface can be parameterized in the same way as a generic class. In the code below, the reference `strNode` has the parameterized type `IMonoLink<String>`. It is assigned the reference value of a node of inferred type `MonoNode<String>`. The assignment is legal, since the parameterized type `MonoNode<String>` is a subtype of the parameterized type `IMonoLink<String>`:

Click here to view code image

```
IMonoLink<String> strNode2 = new MonoNode<>("Bye", null);
System.out.println(strNode2.getData());                    // Prints: Bye
```

As with non-generic interfaces, generic interfaces cannot be instantiated either:

Click here to view code image

```
IMonoLink<String> strNode3 = new IMonoLink<>("Bye", null); // Compile-time err
```

Example 11.4 shows a non-generic class implementing a generic interface. The generic interface `IMonoLink<E>` is parameterized by a *concrete type*, namely, `Lymph`. The type `LymphNode` is a subtype of the parameterized type `IMonoLink<Lymph>`, as it implements the methods of the generic interface `IMonoLink<E>` in accordance with the concrete type parameter `Lymph`.

The Java standard library contains many examples of generic interfaces. The two interfaces `java.lang.Comparable<E>` and `java.util.Comparator<E>` are discussed in detail in §14.4, p. 761, and §14.5, p. 769, respectively. The Java Collections Framework also includes many examples of generic interfaces, such as `Collection<E>`, `List<E>`, `Set<E>`, and `Map<K,V>` (Chapter 15, p. 781).

Example 11.4 *A Non-Generic Class Implementing a Generic Interface*

Click here to view code image

```
// File: LymphNode.java
class Lymph { /*... */ }

public class LymphNode implements IMonoLink<Lymph> {
  private Lymph            body;
  private IMonoLink<Lymph> location;
  @Override public void  setData(Lymph obj) { body = obj; }
  @Override public Lymph getData()          { return body; }
  @Override public void  setNext(IMonoLink<Lymph> loc) { this.location = loc;
  @Override public IMonoLink<Lymph> getNext()          { return this.location;
}
```

Extending Generic Types

A non-final generic type can be extended. Example 11.5 shows that the generic interface `IBiLink<E>` extends the generic interface `IMonoLink<E>`, and that the generic class `BiNode<E>` extends the generic class `MonoNode<E>` and implements the generic interface `IBiLink<E>` (see Figure 11.1).

Click here to view code image

```
interface IBiLink<E> extends IMonoLink<E> {
  // ...
}
```

```
class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  // ...
}
```

The compiler checks that the formal type parameters of the superclass in the `extends` clause can be resolved. In the case above, the formal type parameter `E`, which is specified for the subclass, is also used as the type parameter for the superclass and is used to constrain the interface to the same type parameter. This dependency ensures that an invocation of the subclass will result in the same actual type parameter being used by the superclass and for the interface.

Click here to view code image

```
BiNode<Integer> intBiNode = new BiNode<>(2020, null, null);
MonoNode<Integer> intMonoNode = intBiNode;          // (1)
Integer iRef = intMonoNode.getData();               // Integer with int value 20
MonoNode<Number> numMonoNode = intBiNode;           // (2) Compile-time error!
```

The assignment at (1) is type-safe, as the parameterized class `BiNode<Integer>` is a subtype of the parameterized class `MonoNode<Integer>`. It is important to note that the superclass and the subclass are parameterized with the *same* type parameter; otherwise, the subtype relationship between the superclass and the subclass does not hold. We get a compile-time error at (2) because the parameterized class `BiNode<Integer>` is *not* a subtype of the parameterized class `MonoNode<Number>`. Subtype relationships for generic types are discussed in a later section (p. 579).

Figure 11.1 *Extending Generic Types*

Example 11.5 *Extending Generic Types*

Click here to view code image

```
interface IBiLink<T> extends IMonoLink<T> {
  void        setPrevious(IBiLink<T> previous);
  IBiLink<T> getPrevious();
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E>  previous;    // Reference to previous node

  BiNode(E data, IBiLink<E> next, IBiLink<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  @Override public void setPrevious(IBiLink<E> previous) {
    this.previous = previous;
  }
  @Override public IBiLink<E> getPrevious() { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
           this.getData() +
          (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

Example 11.5 showed examples of generic types being extended to create new generic subtypes. We can extend a non-generic type to a generic subtype as well:

Click here to view code image

```
class AbstractNode { /* ... */ }                      // A non-generic superty
class SimpleNode<E> extends AbstractNode { /* ... */ }// A generic subtype
```

We can also extend concrete parameterized types to specialized non-generic subtypes:

Click here to view code image

```
class IntegerBiNode extends BiNode<Integer> {         // A non-generic subtype
  IntegerBiNode(Integer data, IntegerBiNode next, IntegerBiNode previous) {
    super(data, next, previous);
  }
  //...
}
```

Note that a subtype can inherit only one parameterization of the same generic interface supertype. Implementing or extending a parameterized type fixes the parameterization for the subtype and its supertypes. In the declaration below, the subtype `WeirdNode<E>` tries to implement the interface `IMonoLink<Integer>`, but at the same time, it is a subtype of the interface `IMonoLink<E>` which the superclass `Mono-Node<E>` implements:

Click here to view code image

```
class WeirdNode<E> extends MonoNode<E> implements IMonoLink<Integer> { // Erro
  //...
}
```

There is great flexibility in extending reference types, but care must be exercised to achieve the desired result.

Raw Types and Unchecked Warnings

A generic type without its formal type parameters is called a *raw type*. The raw type is the supertype of all parameterized types of the generic type. For example, the raw type `Node` is the supertype of the parameterized types `Node<String>`, `Node<Integer>`, and `Node<Node<String>>`. The last parameterized type is an example of a *nested parameterization*. It means that a node of this type has a node of type `Node<String>` as data.

A parameterized type (e.g., `Node<String>`) is *not* a class. Parameterized types are used by the compiler to check that objects created are used correctly in the program. The parameterized types `Node<String>`, `Node<Integer>`, and `Node<Node<String>>` are all represented at runtime by their raw type `Node`. In other words, the compiler does *not* create a new class for each parameterized type. Only one class (`Node`) exists that has the name of the generic class (`Node<E>`), and the compiler generates only one class file (`Node.class`) with the Java bytecode for the generic class.

Only reference types (excluding array creation and enumerations) can be used in invocations of generic types. A primitive type is not permitted as an actual type parameter, the reason being that values of primitive types have different sizes. This would require different code being generated for each primitive type used as an actual type parameter, but there is only one implementation of a generic class in Java.

Generics are implemented in the compiler only. The JVM is oblivious about the use of generic types. It does not differentiate between `Node<String>` and `Node<Integer>`, and just knows about the class `Node`. The compiler translates the generic class by a process known as *type erasure*; meaning that information about type parameters is erased and casts are inserted to make the program type-safe at runtime. The compiler guarantees that casts added at compile time never fail at runtime, when the program compiles without any *unchecked warnings*.

It is possible to use a generic class by its raw type only, like a non-generic class, without specifying actual type parameters for its usage. Example 11.6 illustrates mixing generic and non-generic code. The compiler will issue an unchecked warning if such a use can be a potential problem at runtime. Such usage is permitted for backward compatibility with legacy code, but is strongly advised against when writing new code.

The assignment at (5) in Example 11.6 shows that it is always possible to assign the reference value of a parameterized type to a reference of the raw type, as the latter is the supertype of the former. However, the raw type reference can be used to violate the type-safety of the node at runtime, as shown at (6). Calling a method on a node using the raw type reference results in an *unchecked call warning* by the compiler. In this particular case, a `String` is set as the data of an `Integer` node.

Click here to view code image

```
...
Node<Integer> intNode = new Node<>(2020, null);
Integer iRef = intNode.getData();       // Integer object with int value 2020
...
Node rawNode = intNode;             // (5) Assigning to raw type always possible.
rawNode.setData("BOOM");            // (6) Unchecked call warning!
intNode = rawNode;                  // (7) Unchecked conversion warning!
iRef = intNode.getData();           // (8) ClassCastException!
iRef = rawNode.getData();           // (9) Compile-time error!
```

Assigning the reference value of a raw type to a reference of the parameterized type results in an *unchecked conversion warning* from the compiler, as shown at (7). If the node referred to by the raw type reference is not of type `Integer`, using it as a node of type `Integer` can lead to problems at runtime, as shown at (8). The assignment at (8) is only type compatible, not type-safe, as its type-safety is compromised at (6) as explained above. A `ClassCastException` is thrown at runtime, since an `Integer` was expected, but a `String` was returned by the `getData()` method.

The assignment at (9) does not compile because of type mismatch: Without the generic type information, the compiler infers that the call on the `getData()` method using the raw type reference `rawNode` can only return an `Object`, whereas the type of the variable on the left-hand side is `Integer`.

The class `Preliminaries` in Example 11.6 is shown compiled with the non-standard option `-Xlint:unchecked`. The compiler recommends using this option when non-generic and generic code are mixed in this way. The program compiles in spite of the unchecked warnings, and can be executed. But all guarantees of type-safety are off in the face of unchecked warnings. See also §11.11, p. 613, which provides details on translation of generic code by type erasure.

Example 11.6 *Unchecked Warnings*

Click here to view code image

```
// A client for the generic class Node<E> in Example 11.2, p. 568.
public class Preliminaries {
  public static void main(String[] args) {

    Node<Integer> intNode = new Node<>(2018, null);
    Integer iRef = intNode.getData();         // Integer object with int value
    intNode.setData(2020);                    // Ok.
//  intNode.setData("TwentyTwenty");          // (1) Compile-time error!
    intNode.setNext(new Node<>(2019, null)); // (2020, (2019, null))
//  intNode.setNext(new Node<>("Hi", null)); // (2) Compile-time error!
```

```
    Node<String> strNode = new Node<>("Hi", null);
//  intNode = strNode;               // (3) Compile-time error!
    String str = strNode.getData(); // (4) No explicit cast necessary.

    Node rawNode = intNode;          // (5) Assigning to raw type always possib
    rawNode.setData("BOOM");         // (6) Unchecked call warning!
    intNode = rawNode;               // (7) Unchecked conversion warning!
    iRef = intNode.getData();        // (8) ClassCastException!
//  iRef = rawNode.getData();        // (9) Compile-time error!
  }
}
```

Compiling the program:

Click here to view code image

```
>javac -Xlint:unchecked Preliminaries.java
Preliminaries.java:16: warning: [unchecked] unchecked call to setData(E) as a
member of the raw type Node
    rawNode.setData("BOOM");         // (6) Unchecked call warning!

                   ^
  where E is a type-variable:
    E extends Object declared in class Node
Preliminaries.java:17: warning: [unchecked] unchecked conversion
    intNode = rawNode;               // (7) Unchecked conversion warning!

              ^
  required: Node<Integer>
  found:    Node
2 warnings
```

Running the program:

Click here to view code image

```
>java Preliminaries
Exception in thread "main" java.lang.ClassCastException: java.lang.String cann
 be cast to java.lang.Integer
        at Preliminaries.main(Preliminaries.java:18)
```

11.3 Collections and Generics

Before the introduction of generics in Java 1.5, a collection in the Java Collections Framework could hold references to objects of any type. For example, any object which is an instance of the `java.lang.Object` class or its subclasses could be maintained in an instance of the `java.util.ArrayList` class, which implements the `java.util.List` interface.

Click here to view code image

```
List wordList = new ArrayList();     // Using non-generic types.
wordList.add("two zero two zero");   // Can add any object.
wordList.add(2020);
```

```
//...
Object element = wordList.get(0);    // Always returns an Object.
//...
if (element instanceof String str) { // Runtime check to avoid ClassCastExcept
  // Use reference str.
}
```

The client of a collection has to do most of the bookkeeping with regard to using the collection in a type-safe manner: which objects are put in the collection and how objects retrieved are used. Using the `Object` class as the element type allows the implementation of the collection class to be specific, but its use to be generic. An `ArrayList` is a specific implementation of the `List` interface, but usage of the class `ArrayList` is generic with regard to any object.

Using a generic collection, the compiler provides the type-safety, and the resulting code is less verbose.

Click here to view code image

```
List<String> wordList = new ArrayList<>();        // Using a specific type.
wordList.add("two zero two zero");                // Can add strings only.
wordList.add(2020);                               // Compile-time error!
//...
String element = wordList.get(0);                 // Always returns a String.
//...
```

Runtime checks or explicit casts are not necessary now. Generic types allow the implementation of the collection class to be generic, but its use to be specific. The generic type `ArrayList<E>` is a generic implementation of the `List<E>` interface, but now the usage of the parameterized type `ArrayList<String>` is specific, as it constrains the generic type `ArrayList<E>` to strings.

11.4 Wildcards

In this section, we discuss how using wildcards can increase the expressive power of generic types. But first we examine one major difference between array types and parameterized types. The generic class `Node<E>` used in this subsection is defined in Example 11.2, p. 568.

The Subtype Covariance Problem with Parameterized Types

The following three declarations create three nodes of `Integer`, `Double`, and `Number` type, respectively.

Click here to view code image

```
Node<Integer> intNode    = new Node<>(2020,null);          // (1)
Node<Double>  doubleNode = new Node<>(3.14,null);          // (2)
Node<Number>  numNode    = new Node<>(2021, null);         // (3)
```

In the declaration at (3), the signature of the constructor call is `Node(Integer, null)`. The formal type parameter `E` of the generic class `Node<E>` is bound to the actual type parameter `Number`—that is, the signature of the constructor is `Node(Number, Node<Number>)`. Since the type `Integer` is a subtype of the type `Number`, and `null` can be assigned to any reference, the constructor call succeeds.

In the method calls at (4) and (5) below, the method signature in both cases is `setData(Number)`. The method calls again succeed, since the actual parameters are of types `Double` and `Integer`, which are subtypes of `Number`:

Click here to view code image

```
numNode.setData(10.5);                                 // (4)
numNode.setData(2022);                                 // (5)
```

However, the following calls do *not* succeed:

Click here to view code image

```
numNode.setNext(intNode);                              // (6) Compile-time error!
numNode = new Node<Number>(2030, doubleNode);          // (7) Compile-time error!
```

The actual type parameter at (6) is determined to be `Number`. The generic class `Node<E>` is thus parameterized with the class `Number`. The compiler complains that the method `setNext(Node<Number>)` in the parameterized class `Node<Number>` is *not* applicable for the actual argument (`Node<Integer>`) at (6)—that is, the method signature `setNext(Node<Number>)` is *not* compatible with the method call signature `setNext(Node<Integer>)`. The compiler also complains at (7): The constructor signature `Node(Number, Node<Number>)` is *not* applicable for the arguments `(int, Node<Double>)`. The problem is with the second argument at (7). We cannot pass an argument of type `Node<Integer>` or `Node<Double>` where a parameter of type `Node<Number>` is expected. The following assignments will also not compile:

Click here to view code image

```
numNode = intNode;                                       // (8) Compile-time error
numNode = doubleNode;                                    // (9) Compile-time error
```

The reason for the compile-time errors is that `Node<Integer>` and `Node<Double>` are *not* subtypes of `Node<Number>`, although `Integer` and `Double` are subtypes of `Number`. In the case of arrays, the array types `Integer[]` and `Double[]` *are* subtypes of the array type `Number[]`. The subtyping relationship between the individual types carries over to corresponding array types. This type relationship is called *subtype covariance* (see Figure 11.2). This relationship holds for arrays because the element type is available at runtime, and can be checked. If the subtype covariance were allowed for parameterized type, it could lead to problems at runtime, as the element type would not be known and cannot be checked, since it has been erased by the compiler.

Click here to view code image

```
numNode = intNode;                    // If this assignment was allowed,
numNode.setData(25.5);                // the data could be corrupted,
Integer iRef = intNode.getData();     // resulting in a ClassCastException!
```

Therefore, the *subtype covariance* relationship does *not* hold for parameterized types that are instantiations of the same generic type with different actual type parameters, regardless of any subtyping relationship between the actual type parameters. The actual type parameters are *concrete* types (e.g., `Integer`, `Number`), and therefore, the parameterized types are called *concrete parameterized types*. Such parameterized types are totally unrelated. As an example from the Java Collections Framework, the parameterized type `Map<Integer, String>` is not a subtype of the parameterized type `Map<Number, String>`.

Figure 11.2 *No Subtype Covariance for Parameterized Types*

Wildcard Types

Wildcard types are type parameters defined using the *wildcard* symbol `?`. The wildcard `?` by itself represents *all* types. The parameterized type `List<?>` represents a list of all types, whereas the concrete parameterized type `List<Integer>` only represents a list of `Integer`. In other words, a *wildcard type* can represent *many types*. Therefore, a parameterized type that has wildcard types as actual type parameters can represent a family of types, in contrast to a concrete parameterized type that only represents itself. The wildcard types provided in Java represent four subtype relationships that are summarized in Table 11.1.

Wildcard types provide the solution for increased expressive power to overcome the limitations discussed earlier when using generics in Java, but introduce limitations of their own as to what operations can be carried out on an object using references of wildcard types. We will use the class `Node<E>` in Example 11.2, p. 568, as a running example to discuss the use of wildcard types.

Table 11.1 *Summary of Subtyping Relationships for Generic Types*

Name	**Syntax**	**Semantics**	**Description**
Subtype covariance	`? extends Type`	Any subtype of `Type` (including `Type`)	*Bounded wildcard with upper bound*
Subtype contravariance	`? super Type`	Any supertype of `Type` (including `Type`)	*Bounded wildcard with lower bound*
Subtype bivariance	`?`	All types	*Unbounded wildcard*
Subtype invariance	`Type`	Only type `Type`	*Type parameter/argument*

Figure 11.3 *Partial Type Hierarchy for* `Node<? extends Number>`

The wildcard type `? extends Number` denotes all subtypes of `Number`, and the parameterized type `Node<? extends Number>` denotes the family of invocations of `Node<E>` for types that are subtypes of `Number`. Figure 11.3 shows a partial type hierarchy for the parameterized type `Node<? extends Number>`. Note that the parameterized type `Node<? extends Integer>` is a subtype of the parameterized type `Node<? extends Number>`, since the wildcard type `? extends Integer` represents all subtypes of `Integer`, and these are also subtypes of `Number`.

Click here to view code image

```
Node<? extends Integer> intSubNode = new Node<Integer>(100, null);
Node<? extends Number> numSupNode = intSubNode;
```

Subtype Contravariance: `? super Type`

The wildcard type `? super Type` represents all *supertypes* of `Type` (including `Type` itself). The wildcard type `? super Type` is called a *lower bounded wildcard* with `Type` representing its *lower bound*.

Figure 11.4 *Partial Type Hierarchy for* `Node<? super Integer>`

The wildcard type `? super Integer` denotes all supertypes of `Integer`, and the parameterized type `Node<? super Integer>` denotes a family of invocations of `Node<E>` for types that are supertypes of `Integer`. Figure 11.4 shows a partial type hierarchy for the parameterized type `Node<? super Integer>`. Note that the parameterized type `Node<? super Number>` is a *subtype* of the parameterized type `Node<? super Integer>`, since the wildcard type `? super Number` represents all supertypes of `Number`, and these are also supertypes of `Integer`.

Click here to view code image

```
Node<? super Number> numSupNode = new Node<Number>(100, null);
Node<? super Integer> numIntSupNode = numSupNode;
```

Subtype Bivariance: `?`

As mentioned earlier, the wildcard type `?` represents *all* types. The wildcard type `?` is called the *unbounded wildcard*, since it has no bounds as do the other two wildcard types. By definition, it represents both the upper and the lower bounded wildcards for any bound.

The parameterized type `Node<?>` denotes the family of invocations of `Node<E>` for any type—that is, denotes a `Node` of any kind, and is therefore the supertype of all invocations of `Node<E>` (see also Figure 11.5, p. 583, and §11.5, p. 584).

Figure 11.5 *Partial Type Hierarchy for Selected Parameterized Types of* `Node<E>`

Subtype Invariance: `Type`

When a concrete type `Type` is used as an actual type parameter in a parameterized type, it represents `Type` itself. Since `Type` can be *any* concrete type, it is called an *unbounded type parameter*. The concrete parameterized type `Node<Integer>` represents the invocation of `Node<E>` for the concrete actual type parameter `Integer`. As we have seen earlier, there is no subtype covariance relationship between concrete parameterized types, but there is such a relationship between bounded parameterized types and concrete parameterized types (see also Figure 11.3 and Figure 11.4).

Let us recapitulate the basic terminology before proceeding further. A generic type can specify one or more formal type parameters. A parameterized type is an invocation of a generic type, supplying the required actual type parameters. An actual type parameter can be a wildcard type (possibly bounded) or a concrete type. A concrete type is either a non-generic type or a parameterized type that has concrete types as parameters.

Some Restrictions on Wildcard Types

Wildcards *cannot* be used in instance creation expressions:

Click here to view code image

```
Node<?> anyNode = new Node<?>(2020, null);                  // Compile-time err
Node<? extends Integer> extIntNodeA
               = new Node<? extends Integer>(0, null);      // Compile-time err
Node<? extends Integer> extIntNodeB = new Node<Integer>(0, null);  // OK
```

The actual type parameter in the constructor call must be a concrete type. Creating instances of wildcard parameterized types is analogous to instantiating interface types; neither can be used in object creation expressions.

Wildcards *cannot* be used in the *header* of reference type declarations. Supertypes in the `extends` and `implements` clauses cannot have wildcards.

Click here to view code image

```
class QuestionableNode<?>  { /* ... */ }                              // Not OK
class SubNode extends Node<?> { /* ... */ }                           // Not OK
interface INode extends Comparable<? extends Node<?>> { /* ... */ }  // Not OK
class XNode implements Comparable<?>  { /* ... */ }                   // Not OK
```

However, *nested wildcards* are not a problem in a reference type declaration header or in an object creation expression:

Click here to view code image

```
class OddNode extends Node<Node<?>> implements Comparable<Node<?>> { /* ... */
...
Node<?> nodeOfAnyNode = new Node<Node<?>>(new Node<Integer>(2020, null), null)
```

11.5 Using References of Wildcard Parameterized Types

A wildcard type can be used to declare parameterized references—that is, references whose type is a wildcard parameterized type. In this section, we look at how such references are used in the following contexts:

- Assignment between such references
- Calling methods of generic types using such references

The generic class `Node<E>` used in this subsection is defined in Example 11.2, p. 568.

Generic Reference Assignment

A reference of a supertype can refer to an object of a subtype, and this substitution principle applies to parameterized types as well. Assignment compatibility is according to the type hierarchy of the parameterized types. Figure 11.5 shows partial type hierarchy for selected parameterized types of the generic class `Node<E>`. It combines the type hierarchies from Figure 11.3 and Figure 11.4. As we would expect, *widening reference conversions* according to the type hierarchy are always type-safe. All but the last assignment statement in the code below are legal. The types `Node<Number>` and `Node<Integer>` are unrelated. (The notation `B <: A` means `B` is a subtype of `A`.)

Click here to view code image

```
Node<Object>  objNode = new Node<Object>(100, null);
Node<Number>  numNode = new Node<Number>(200, null);
Node<Integer> intNode = new Node<Integer>(300, null);
Node<? extends Number> extNumNode
                       = intNode; // Node<Integer> <: Node<? extends Number>
Node<? super Integer>  supIntNode
                       = numNode; // Node<Number> <: Node<? super Integer>
supIntNode = objNode;             // Node<Object> <: Node<? super Integer>
numNode    = intNode;             // Compile-time error! Types unrelated.
```

In the code below, we get an error at (1) because the types `Node<? extends Number>` and `Node<? super Number>` are unrelated, but that is not the case for the types `Node<? extends Object>` and `Node<? super Object>` at (2). The family of types denoted by the type `Node<? super Object>` has the subtype `Node<Object>` only, which is also a subtype of the type `Node<? extends Object>`. In the assignment at (3), the type `Node<? extends Object>` is not a subtype of the type `Node<? super Object>`, but the converse is true as established at (2).

Click here to view code image

```
Node<? super Number>   supNumNode;
Node<? extends Object> extObjNode;
Node<? super Object>   supObjNode;
extNumNode = supNumNode;  // (1) Compile-time error! Types unrelated.
extObjNode = supObjNode;  // (2) Node<? super Object> <: Node<? extends Object
supObjNode = extObjNode;  // (3) Compile-time error!
```

Narrowing reference conversion requires an explicit cast, except for the cases noted below (see also Figure 11.5). The raw type `Node` and the unbounded wildcard parameterized type `Node<?>` are essentially equivalent in this regard. Conversion between the two is type-safe:

Click here to view code image

```
Node    rawNode;
```

```
Node<?> anyNode;
rawNode = anyNode;  // Node <-- Node<?> is type-safe.
anyNode = rawNode;  // Node<?> <-- Node is type-safe.
```

The unbounded wildcard parameterized type `Node<?>` and the upper bounded wildcard parameterized type `Node<? extends Object>` are also essentially equivalent (see (4)), except when assigned a value of the raw type `Node` (see (5)).

Click here to view code image

```
// (4):
anyNode = extObjNode;  // Node<?> <-- Node<? extends Object> is type-safe.
extObjNode = anyNode;  // Node<? extends Object> <-- Node<?> is type-safe.

// (5):
anyNode    = rawNode;  // Node<?> <-- Node is type-safe.
extObjNode = rawNode;  // Node<? extends Object> <-- Node: Unchecked Conversio
```

Assigning a value of the raw type `Node` to a reference of the type `Node<? extends Object>` results in an *unchecked conversion warning*—which conforms to the general rule when mixing legacy and generic code: Assigning the value of a raw type to a reference of a bounded wildcard parameterized type or a concrete parameterized type results in an unchecked conversion warning, as illustrated by the examples below.

Click here to view code image

```
extNumNode = rawNode; // Node<? extends Number> <-- Node: Unchecked Conversion
intNode    = rawNode; // Node<Integer> <-- Node: Unchecked Conversion
```

For a discussion of explicit casting of parameterized references, see §11.13, p. 625. Suppressing different kinds of unchecked warnings with the `@SuppressWarnings("unchecked")` annotation is discussed in §11.13, p. 623, and §25.5, p. 1582.

Using Parameterized References to Call Set and Get Methods

Generic classes are suitable for implementing ADTs called *collections* (also called *containers*) where the element type is usually specified by a type parameter. The Java Collections Framework is a prime example of such collections. A collection usually provides two basic operations: a *set* operation (also called a *write* or *put* operation) to add an element to the collection, and a *get* operation (also called a *read* operation) to retrieve an element from the collection. The *set* operation takes a parameter of the type `T`, where `T` is a type parameter of the generic class. The *get* operation returns a value of the type parameter `T`. The class `Node<E>` provides these two basic operations to manipulate the data in a node:

Click here to view code image

```
class Node<E> {
  private E data;
  // ...
  public void setData(E obj) { data = obj; }          // (1) Set operation.
  public E    getData()      { return data; }         // (2) Get operation.
  // ...
}
```

So far we have called these two methods using references of concrete parameterized types:

Click here to view code image

```
Node<Number> numNode = new Node<>(2020, null);
numNode.setData(2021);                          // (3) Can only set a Number.
Number data = numNode.getData();                // (4) Can only get a Number.
```

The actual type parameter in the above method calls is a *concrete type*, but what happens when we use a reference of a wildcard parameterized type that represents a family of types? For example, what if the type of the reference `numNode` is `Node<? extends Number>`? Is the method call at (3) type-safe? Is the assignment at (4) type-safe? Operations that can potentially break the type-safety are flagged as either compile-time errors or warnings. If there are warnings but no errors, the program still compiles. However, type-safety at runtime is not guaranteed.

The key to using generics in Java is understanding the implications of wildcard parameterized types in the language—and why the compiler will not permit certain operations involving wildcards, since these might break the type-safety of the program. To illustrate some of the subtleties, we compile the class in Example 11.7 successively with different headers for the method `checkIt()`. The parameter type is different in each method header, from (h1) to (h5). The method uses three local variables `object`, `number`, and `integer` of type `Object`, `Number`, and `Integer`, respectively ((v1) to (v3)). There are three calls to the `setData()` method of the generic class `Node<E>` to set an `Object`, a `Number`, and an `Integer` as the data in the node referenced by the reference `s0` ((s1) to (s3)). There are also three calls to the `getData()` method of the generic class `Node<E>`, assigning the return value to each of the local variables ((s4) to (s6)). And finally, the last statement, (s7), tests whether the data retrieved can be put back in again.

Example 11.7 *Illustrating Get and Set Operations Using Parameterized References*

Click here to view code image

```
class WildcardAccessRestrictions {

  static void checkIt(Node s0) {                              // (h1)
//static void checkIt(Node<?> s0) {                           // (h2)
//static void checkIt(Node<? extends Number> s0) {            // (h3)
//static void checkIt(Node<? super Number> s0) {              // (h4)
//static void checkIt(Node<Number> s0) {                      // (h5)
    // Local variables
    Object  object  = new Object();                           // (v1)
    Number  number  = 1.5;                                    // (v2)
    Integer integer = 10;                                     // (v3)
    // Method calls
    s0.setData(object);                                       // (s1)
    s0.setData(number);                                       // (s2)
    s0.setData(integer);                                      // (s3)
    object  = s0.getData();                                   // (s4)
    number  = s0.getData();                                   // (s5)
    integer = s0.getData();                                   // (s6)
    s0.setData(s0.getData());                                 // (s7)
  }
}
```

Attempts to compile the method in Example 11.7 with different headers are shown in Table 11.2. The rows are statements from (s1) to (s7) from Example 11.7. The columns indicate the type of the parameter `s0` in the method headers (h1) to (h5). The reference `s0` is used to call the methods. The entry **ok** means the compiler did not report any errors or any unchecked warnings. The entry **!** means the compiler did not report any errors but issued an unchecked call warning. The entry × means the compiler reported an error. In other words, we cannot carry out the operations that are marked with the entry ×.

Table 11.2 *Get and Set Operations Using Parameterized References*

Type of Operation					
`s0.setData(object);`	**!** ×				
`s0.setData(number);`	**!** **ok**				
`s0.setData(integer);`	**!** **ok**				
`object = s0.getData();`	**ok**				
`number = s0.getData();`	**ok**				
`integer = s0.getData();`	×				
`s0.setData(s0.get-Data());`	**!** **ok**				

Raw Type References

The type of the reference `s0` is the raw type `Node`. This case illustrates the non-generic paradigm of using a collection: We can put any object, but we can only get an `Object`. From Table 11.2, we see that we can put any object as data in a node of the raw type `Node`, but the compiler issues *unchecked call warnings*, as we are putting an object into a raw type node whose element type is not known. We can only get an `Object`, as we cannot be more specific about the data type.

Unbounded Wildcard References: `<?>`

The type of the reference `s0` is `Node<?>`. The compiler determines that the actual type parameter for each method call is the wildcard `?`—that is, any type. Obviously, we cannot set any data in a node whose element type cannot be determined. It might not be type-safe. And we cannot guarantee the type of its data either because the data type can be of any type, but we can safely read it as an `Object`. Note that we can always write a `null`, as the `null` value can be assigned to any reference.

Typical use of unbounded wildcard reference is in writing methods that treat the objects in a container as of type `Object` and make use of `Object` methods for their operation. The method below can print the data in *any* sequence of nodes, given the start node. The specific type of the data in the node is immaterial. Note also that the method is not dependent on any type parameter.

Click here to view code image

```
void printNodeDataSequence(Node<?> s0) {        // Unbounded parameterized type
  Node<?> next = s0.getNext();                  // Returns node as Node<Object>
  while (next != null) {
    Object obj = next.getData();                // Object reference.
    System.out.println(obj.toString());         // Call Object method
    next = next.getNext();
  }
}
```

Upper Bounded Wildcard References: `<? extends Type>`

The type of the reference `s0` is `Node<? extends Number>`, where `Type` is `Number`. This means that the reference `s0` refers to a node containing an object whose type is either `Number` or a subtype of `Number`, but the specific (sub)type of the object cannot always be determined at compile time. Putting *any* object, except a `null`, into such a node might not be type-safe.

The code below shows what would happen if any object was allowed to be set as data in a `Long` node via its alias `s0`. If (1), (2), or (3) were allowed, we would get a `ClassCastException` at (4) because the data could not be assigned to a `Long` reference, as the type-safety of the node `longNode` will have been compromised, either with a supertype object or an object of an unrelated type.

Click here to view code image

```
Long longInt = 20L;
Node<Long> longNode = new Node<>(longInt, null);  // Node of Long, that is
Node<? extends Number> s0 = longNode;// referenced by a Node<? extends Number>
s0.setData(object);                  // If this was allowed, or             (1
s0.setData(number);                  // if this was allowed, or             (2
s0.setData(integer);                 // if this was allowed,                (3
longInt = longNode.getData();        // we would get an exception here.     (4
```

The following method call will also not compile, as the compiler cannot give any guarantees at compile time that the reference `s0` will refer to a node of `Long` at runtime:

Click here to view code image

```
s0.setData(longInt);                 // Compile-time error!
```

The upper bound in the wildcard type `? extends Number` is `Number`. Therefore, the data of the node with the wildcard type `? extends Number` must be a `Number` (i.e., either an object of type `Number` or an object of a subtype of `Number`). Thus we can only safely assign the reference value returned by the *get* operation to a reference of type `Number` or a supertype of `Number`.

Lower Bounded Wildcard References: `<? super Type>`

Using a reference of type `Node<? super Number>`, where `Type` is `Number`, we can only put a `Number` or a subtype object of `Number` into the node, as such a number would also be a subtype object of any supertype of `Number`. Since we cannot guarantee which specific supertype of `Number` the node actually has, we cannot put any supertype object of `Number` in the node. The code below shows what would happen if an unrelated supertype object was put in as data in a `Node<? super Number>`. If (1) were allowed, we would get a `ClassCastException` at (2) because the data value (of a supertype) cannot be assigned to a `Number` reference (which is a subtype).

Click here to view code image

```
Node<Number> numNode = new Node<>(2020, null);
Node<? super Number> s0 = numNode;
s0.setData(object);             // (1) If this set operation was allowed,
number = numNode.getData();     // (2) we would get an exception here.
```

Since the type of the reference `s0` is `Node<? super Number>`, the reference `s0` can refer to a node containing an object whose type is either `Number` or some supertype of `Number`. When we *get* the data from such a node, we can only safely assume that it is an `Object`. Keeping in mind that a reference of a supertype of `Number` can refer to objects that are unrelated to `Number` (e.g., an `Object` reference that can refer to a `String`), if (3) were allowed in the code below, we would get a `ClassCastException` at (3):

Click here to view code image

```
Node<Object> objNode = new Node<>("Hi", null);  // String as data.
Node<? super Number> s0 = objNode;
number = s0.getData();         // (3) If allowed, we would get an exception her
object = s0.getData();         // This is always ok.
```

Unbounded Type References: `<Type>`

The type of the reference `s0` is `Node<Number>`, where `Type` is `Number`. The actual type parameter for each method call is determined to be `Number`. Thus the type of the parameter in the `setData()` method and the return value of the `getData()` method is `Number`. Therefore, we can pass the reference value of a `Number` or a subclass object of `Number` to the `setData()` method, and can assign the reference value returned by the `getData()` method to a reference of the type `Number` or a supertype of `Number`. In this case, we can put a `Number`, and get a `Number`.

Table 11.3 gives a summary of using parameterized references for *set* and *get* operations on a container. Here are some general guidelines for choosing a wildcard parameterized type:

- If we only want to *get* an element of type `E` from a container, we can use the upper bounded wildcard `? extends E` for the reference.
- If we only want to *put* an element of type `E` into a container, we can use the lower bounded wildcard `? super E` for the reference.
- If we want to both *get* and *set* elements of type `E` in a container, we can use the unbounded type `E` for the reference.

The following acronym might help to remember which parameterized references should be used to invoke *get* and *set* methods of a container: *GESS* (*Get*-`extends`-*Set*-`super`), meaning for a *get* method use reference of type `<? extends T>` and for a *set* method use reference of type `<? super T>`.

Table 11.3 *Summary of Get and Set Operations Using Parameterized References*

Type of Operation	**`Node<? extends Number>`** **`Node<? super Number>`**				
Get/read `Number` Any object	*Cannot put anything except* `nulls`. *Number or its subtypes*				
Set/put `Number` `Object` *only*					

11.6 Bounded Type Parameters

In the declaration of the generic class `Node<E>`, the type parameter `E` is *unbounded*— that is, it can be any reference type. However, sometimes it may be necessary to restrict what type the type parameter `E` can represent. The canonical example is restricting that the type parameter `E` is `Comparable<E>` so that objects can be compared.

Wildcard types *cannot* be used in the header of a generic class to restrict the type parameter:

Click here to view code image

```
class CmpNode<? extends Comparable> { ... }          // Compile-time error!
```

However, the type parameter can be bounded by a *constraint* as follows:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> {             // E is bounded.
  // ...
}
```

In the constraint `<E extends Comparable<E>>`, `E` is *bounded* and `Comparable<E>` is the *upper bound*. This is an example of a *recursive type bound*. The declaration above states that the actual type parameter when we parameterize the generic class `CmpNode` must implement the `Comparable` interface, and that the objects of the actual type parameter must be comparable to each other. This implies that the type, say `A`, that we can use to parameterize the generic class, must implement the parameterized interface `Comparable<A>`.

If we base the implementation of the `CmpNode` class on the generic class `Node<E>`, we can write the declaration as follows:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E> {
  // ...
}
```

The `extends` clause is used in two different ways: for the generic class `CmpNode` to extend the class `Node<E>`, and to constrain the type parameter `E` of the generic class `CmpNode` to the `Comparable<E>` interface. Although the type parameter `E` must implement the interface `Comparable<E>`, we do *not* use the keyword `implements` in a constraint. Neither can we use the `super` clause to constrain the type parameter of a generic class.

If we want `CmpNode`s to have a natural ordering based on the natural ordering of their data values, we can declare the generic class `CmpNode` as shown in Example 11.8:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {
 // ...
}
```

Note how the `Comparable` interface is parameterized in the `implements` clause. The constraint `<E extends Comparable<E>>` specifies that the type parameter `E` is `Comparable`, and the clause `implements Comparable<CmpNode<E>>` specifies that the generic class `CmpNode` is `Comparable`.

Example 11.8 *Implementing the* `Comparable<E>` *Interface*

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override public int compareTo(CmpNode<E> node2) {
    return this.getData().compareTo(node2.getData());
  }
}
```

Here are some examples of how the generic class `CmpNode` in Example 11.8 can be parameterized:

Click here to view code image

```
CmpNode<Integer> intCmpNode  = new CmpNode<>(2020, null);          // (1)
CmpNode<Number>  numCmpNode  = new CmpNode<Number>(2020, null);   // (2) Error
CmpNode<Integer> intCmpNode2 = new CmpNode<>(2021, null);
```

```
int result = intCmpNode.compareTo(intCmpNode2);
```

The actual type parameter `Integer` at (1) implements `Comparable<Integer>`, but the actual type parameter `Number` at (2) is not `Comparable`. In the declaration `CmpNode<A>`, the compiler ensures that `A` implements `Comparable<A>`.

Multiple Bounds

A bounded type parameter can have *multiple bounds*, B_1 `&` B_2 `& ... &` B_n, which must be satisfied by the *actual* type parameter:

Click here to view code image

```
class CmpNode<E extends Number & Serializable> ...
```

An extra bound, the `Serializable` interface, has been added using the ampersand (`&`). The formal type parameter `E` is a subtype of *both* `Number` and `Serializable`, and represents both of these concrete types in the body of the generic class. The constraint above will only allow the generic type to be parameterized by an actual type parameter which is a subtype of *both* `Number` and `Serializable`.

We can add as many bounds as necessary. A type parameter `E` having multiple bounds is a *subtype* of all of the types denoted by the individual bounds. A bound can be a parameterized type, as in the following generic class header:

Click here to view code image

```
class CmpNode<E extends Comparable<E> & Serializable> ...
```

If the raw type of a bound is a (non-final) superclass of the bounded type parameter, it can only be specified as the first bound, and there can only be one such bound (as a subclass can only extend one immediate superclass). The raw type of an individual bound cannot be used with different type arguments, since a type parameter cannot be the subtype of more than one bound having the same raw type. In the class header below, whatever `E` is, it cannot be a subtype of two parameterizations of the same interface type (i.e., `Comparable`) at the same time:

Click here to view code image

```
class CmpNode<E extends Comparable<E> & Serializable & Comparable<String>> //E
```

If the type parameter has a bound, methods of the bound can be invoked on instances of the type parameter in the generic class. Otherwise, only methods from the `Object` class can be invoked on instances of the type parameter. In the declaration of the generic class `Node<E>` in Example 11.2, p. 568, we cannot call any methods on instances of the type parameter except for those in the `Object` class because the type parameter is unbounded. Since the instances of the type parameter `E` are guaranteed to be `Comparable<E>` in the generic class `CmpNode`, we can call the method `com-pareTo()` of the `Comparable` interface on these instances.

11.7 Generic Methods and Constructors

We first look at how generic methods and constructors are declared, and then at how they can be called—both with and without explicit actual type parameters.

Declaring Generic Methods

A generic method (also called *polymorphic method*) is implemented like an ordinary method, except that one or more formal type parameters are specified immediately preceding the return type. In the case of a generic constructor, the formal parameters are specified before the class name in the constructor header. Much of what applies to generic methods in this regard also applies to generic constructors.

Example 11.9 *Declaring Generic Methods*

Click here to view code image

```
public class Utilities {

  // The key type and the array element type can be any type.
  static boolean containsV1(Object key, Object[] array) { // (1) Non-generic
                                                          //     version
    for (Object element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type and the array element type are the same.
  static <E> boolean containsV2(E key, E[] array) {       // (2) Generic versi
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type is a subtype of the array element type.
  static <K extends E, E> boolean containsV3(K key, E[] array) {  // (3)
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
    }
  }
```

In Example 11.9, the method `containsV1()` at (1) is a non-generic method to determine the membership of an arbitrary key in an arbitrary array of objects.

Click here to view code image

```
static boolean containsV1(Object key, Object[] array) { // (1) Non-generic ver
  // ...
}
```

The method declaration at (1) is too general, in the sense that it does not express any relationship between the key and the array. This kind of type dependency between parameters can be achieved by using generic methods. In Example 11.9, the method `containsV2()` at (2) is a generic method to determine the membership of a key of type `E` in an array of type `E`. The type `Object` at (1) has been replaced by the type parameter `E` at (2), with the formal type parameter `E` being specified before the return type, in the same way as for a generic type.

Click here to view code image

```
static <E> boolean containsV2(E key, E[] array) {        // (2) Generic version
  // ...
}
```

As with the generic types, a formal type parameter can have a bound, which is a type (i.e., not a type parameter). A formal type parameter can be used in the return type, in the formal parameter list, and in the method body. It can also be used to specify *bounds* in the formal type parameter list.

A generic method need not be declared in a generic type. If declared in a generic type, a *generic instance method* can also use the type parameters of the generic type as any other non-generic instance methods of the generic type. In contrast, a *generic static method* can only use the type parameters declared in its method header.

Calling Generic Methods

Consider the following class declaration:

Click here to view code image

```
public class ClassDecl {
  static <E_1,..., E_k> void genericMethod(P_1 p_1,..., P_m p_m) { ... }
  // ...
}
```

Note that in the method declaration above, a type `P_i` may or may not be from the list of type parameters `E_1`, ..., `E_k`. We can call the method in various ways. One main difference from calling a non-generic method is that the actual type parameters can be specified before the method name in the call to a generic method. In the method calls shown below, `<A_1,...,  A_k>` are the actual type parameters and `(a_1,...,  a_m)` are the actual arguments. The specification `<A_1,...,  A_k>` of the actual type parameters is known as a *type witness*, as it corroborates the types to use in the method call. If included, it must be specified in its entirety. If there is not type witness, then the compiler *infers* the actual type parameters.

The following method calls can occur in any static or non-static context where the class `CallDecl` is accessible:

Click here to view code image

```
CallDecl ref;
ref.<A_1,..., A_k>genericMethod(a_1,..., a_m);
CallDecl.<A_1,..., A_k>genericMethod(a_1,..., a_m);
```

The following method calls can only occur in a non-static context of the class `CallDecl`:

Click here to view code image

```
this.<A_1,..., A_k>genericMethod(a_1,..., a_m);          // Non-static conte
super.<A_1,..., A_k>genericMethod(a_1,..., a_m);         // Non-static conte
CallDecl.super.<A_1,..., A_k>genericMethod(a_1,..., a_m);  // Non-static conte
```

Another difference from calling non-generic methods is that, if the type witness is explicitly specified, the syntax of a generic *static* method call requires an explicit reference or the raw type. When the type witness is not explicitly specified, the syntax of a generic method call is similar to that of a non-generic method call.

Click here to view code image

```
<A_1,..., A_k>genericMethod(a_1,..., a_m);      // Compile-time error!
genericMethod(a_1,..., a_m);                    // Ok.
```

Here are some examples of calls to the `containsV2()` method at (2) in the class `Utilities` in Example 11.9, where the type witness is specified. We can see from the method signature and the method call signature that the method can be applied to the arguments at

(1), (2), and (3), but not at (4). At (5), we must specify a reference or the class name because a type witness with the actual type parameter is specified.

Click here to view code image

```
Integer[] intArray = {10, 20, 30};

boolean f1 = Utilities.<Integer>containsV2(20, intArray);             // (1) tru
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f2 = Utilities.<Number>containsV2(30.5, intArray);            // (2) fal
// E is Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f3 = Utilities.<Comparable<Integer>> containsV2(20, intArray); // (3)
// E is Comparable<Integer>.
// Method signature:      containsV2(Comparable<Integer>, Comparable<Integer>[
// Method call signature: containsV2(Integer,             Integer[])

boolean f4 = Utilities.<Integer>containsV2(30.5, intArray);           // (4) Err
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Double,  Integer[])

// Requires explicit reference or raw type.
boolean f5 = <Integer>containsV2(20, intArray);                // (5) Syntax err
```

Here are some examples of method calls where the compiler infers the actual type parameters from the method call. At (6), both the key and the element type are `Integer`, the compiler infers that the actual type parameter is `Integer`. At (7), where the key type is `Double` and the element type is `Integer`, the compiler infers the actual type parameter to be `Number`—that is, the first common supertype of `Double` and `Integer`. At (8), the compiler infers the actual type parameter to be `Serializable`— that is, the first common supertype of `String` and `Integer`. In all the cases below, the method is applicable to the arguments.

Click here to view code image

```
boolean f6 = Utilities.containsV2(20, intArray);                      // (6) tru
// E is inferred to be Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f7 = Utilities.containsV2(30.5, intArray);                    // (7) fal
// E is inferred to be Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f8 = Utilities.containsV2("Hi", intArray);                    // (8) fal
// E is inferred to be Serializable.
// Method signature:      containsV2(Serializable, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

At (8), if we had specified the actual type parameter explicitly to be `Integer`, the compiler would flag an error, as shown at (9), since the method signature is not applicable to the arguments:

Click here to view code image

```
boolean f9 = Utilities.<Integer>containsV2("Hi", intArray);         // (9) Err
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(String, Integer[])
```

We can explicitly specify the key type to be a subtype of the element type by introducing a new formal parameter and a bound on the key type, as for the method `containsV3()` at (3) in Example 11.9:

Click here to view code image

```
static <K extends E, E> boolean containsV3(K key, E[] array) {
  // ...
}
```

The following calls at (10) and (11) illustrates inferring of actual type parameters from the method call when no type witness is specified. At (10), the compiler infers the `K` type parameter to be `Double` and the type parameter `E` to be `Number`—that is, the first common supertype of `Double` and the array element type `Integer`. The constraint is satisfied and the method signature is applicable to the arguments.

Click here to view code image

```
boolean f10 = Utilities.containsV3(30.5, intArray);                 // (10) fa
// K is inferred to be Double. E is inferred to be Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Double, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f11 = Utilities.containsV3("Hi", intArray);                 // (11) fa
// K is inferred to be String. E is inferred to be Serializable.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(String, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

At (11), the compiler infers the `K` type parameter to be `String` and the type parameter `E` to be `Serializable`—that is, the first common supertype of `String` and the array element type `Integer`. The constraint is satisfied and the method signature is applicable to the arguments, as both `String` and `Integer` are `Serializable`.

The examples below illustrate how constraints come into play in method calls. At (12), the constraint is satisfied and the method signature is applicable to the arguments. At (13), the constraint is not satisfied; therefore, the call is rejected. At (14), the constraint is satisfied, but the method signature is not applicable to the arguments. The call at (15) is rejected because the number of actual type parameters specified in the call is incorrect.

Click here to view code image

```
boolean f12 = Utilities.<Number, Number>containsV3(30.0, intArray); // (12) fa
// K is Number. E is Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Number, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f13 = Utilities.<Number, Integer>
                        containsV3(30.5, intArray);                 // (13) Er
// K is Number. E is Integer.
// The constraint (K extends E) is not satisfied.

boolean f14 = Utilities.<Integer, Number>
                        containsV3(30.5, intArray);                 // (14) Er
// K is Integer. E is Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Integer, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f15 = Utilities.<Number>containsV3(30.5, intArray);         // (15) Er
// Incorrect no. of type parameters.
```

Typically, the dependencies among the parameters of a method and its return type are expressed by formal type parameters. Here are some examples:

Click here to view code image

```
public static <K,V> Map<V,List<K>> toMultiMap(Map<K,V> origMap) { ... } // (1
public static <N> Set<N> findVerticesOnPath(Map<N,Collection<N>> graph,
                                            N startVertex)      { ... } // (1
```

The method header at (16) expresses the dependency that the map returned by the method has the values of the original map as keys, and its values are lists of keys of the original map—that is, the method creates a *multimap*. In the method header at (17), the type parameter `N` specifies the element type of the set of vertices to be returned, the type of the keys in the map, the element type of the collections that are values of the map, and the type of the start vertex.

11.8 Implementing a Simplified Generic

Stack

The `Node<E>` class from Example 11.2, p. 568, can be used to implement linked data structures. Example 11.10 is an implementation of a simplified generic stack using the `Node<E>` class. The emphasis is not on how to develop a full-blown, industrial-strength implementation, but on how to present a simple example in the context of this book in order to become familiar with code that utilizes generics. For thread-safety issues concerning a stack, see §22.4, p. 1396.

The class `MyStack<E>` implements the interface `IStack<E>` shown in Example 11.10, and uses the class `Node<E>` from Example 11.2. The class `NodeIterator<E>` in Example 11.10 provides an iterator to iterate over linked nodes. The class `MyStack<E>` is `Iterable<E>`, meaning we can use the `for(:)` loop to iterate over a stack of this class (see (9) and (12)). It is instructive to study the code to see how type parameters are used in various contexts, how the iterator is implemented, and how we can use the `for(:)` loop to iterate over a stack. For details on the `Iterable<E>` and `Iterator<E>` interfaces, see §15.2, p. 791.

Example 11.10 *Implementing a Simplified Generic Stack*

Click here to view code image

```
/** Interface of a generic stack */
public interface IStack<E> extends Iterable<E> {
  void push(E element);           // Add the element to the top of the stack
  E pop();                        // Remove the element at the top of the stac
  E peek();                       // Get the element at the top of the stack.
  int size();                     // No. of elements on the stack.
  boolean isEmpty();              // Determine if the stack is empty.
  boolean isMember(E element);    // Determine if the element is in the stack.
  E[] toArray(E[] toArray);       // Copy elements from stack to array
  @Override
  String toString();              // Return suitable text representation of
                                  // elements on the stack: (e1, e2, ..., en)
}
```

Click here to view code image

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/** Simplified implementation of a generic stack */
public class MyStack<E> implements IStack<E> {                     // (1)
  // Top of stack.
  private Node<E> tos;                                             // (2)
  // Size of stack
  private int numOfElements;                                       // (3)

  @Override public boolean isEmpty() { return tos == null; }       // (4)
  @Override public int size()         { return numOfElements; }    // (5)
  @Override public void push(E element) {                        // (6)
    tos = new Node<>(element, tos);
    ++numOfElements;
  }

  @Override public E pop() {                                      // (7)
    if (!isEmpty()) {
      E data = tos.getData();
      tos = tos.getNext();
      --numOfElements;
      return data;
    }
    throw new NoSuchElementException("No elements.");
  }

  @Override public E peek()  {                                    // (8)
    if (!isEmpty()) return tos.getData();
    throw new NoSuchElementException("No elements.");
  }

  // Membership
  @Override public boolean isMember(E element) {                  // (9)
    for (E data : this)
      if (data.equals(element))
        return true;       // Found.
    return false;          // Not found.
  }

  // Get iterator.
  @Override public Iterator<E> iterator() {                       // (10)
    return new NodeIterator<>(this.tos);
  }

  // Copy to array as many elements as possible.
  @Override public E[] toArray(E[] toArray) {                     // (11)
    Node<E> thisNode = tos;
    for (int i = 0; thisNode != null && i < toArray.length; i++) {
      toArray[i] = thisNode.getData();
      thisNode = thisNode.getNext();
    }
    return toArray;
  }

  // Text representation of stack: (e1, e2, ..., en).
  @Override public String toString() {                            // (12)
    StringBuilder rep = new StringBuilder("(");
    for (E data : this) {
      rep.append(data + ", ");
    }
    if (!isEmpty()) {
      int len = rep.length();
      rep.delete(len - 2, len);                      // Delete the last ", ".
    }
    rep.append(")");
    return rep.toString();
  }
}
```

Click here to view code image

```
import java.util.Iterator;

/** Iterator for nodes */
public class NodeIterator<E> implements Iterator<E> {
  private Node<E> thisNode;

  public NodeIterator(Node<E> first) { thisNode = first;  }

  @Override public boolean hasNext() { return thisNode != null; }

  @Override public E next() {
    E data = thisNode.getData();
    thisNode = thisNode.getNext();
    return data;
  }

  @Override public void remove() { throw new UnsupportedOperationException();
}
```

Review Questions

11.1 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_50 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();     // (1)
    intList.add(2020);
    intList.add(2021);
    List<Number> numList = intList;                // (2)
    for (Number n : numList)                       // (3)
      System.out.println(n + " ");
  }
}
```

Select the one correct answer.

a. The code will fail to compile because of an error at (1).

b. The code will fail to compile because of an error at (2).

c. The code will fail to compile because of an error at (3).

d. The code will compile. When run, it will throw a `ClassCastException` at (3).

e. The code will compile and execute normally.

11.2 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_40 {
  public static void main(String[] args) {
    List <? super Integer> sList = new ArrayList<Number>(); //(1)
    int i = 2020;
    sList.add(i);
    sList.add(++i);                                         //(2)
    Number num = sList.get(0);                              //(3)
  }
}
```

11.3 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_70 {
  public static void main(String[] args) {
    List<Integer> glst1 = new ArrayList();      //(1)
    List nglst1 = glst1;                                //(2)
    List nglst2 = nglst1;                               //(3)
    List<Integer> glst2 = glst1;                        //(4)
  }
}
```

Select the one correct answer.

a. The code will compile without any warnings.

b. The code will compile with an unchecked warning at (1).

c. The code will compile with an unchecked warning at (2).

d. The code will compile with an unchecked warning at (3).

e. The code will compile with an unchecked warning at (4).

f. The code will fail to compile.

11.4 Which occurrences of the type parameter `T` are illegal in the following class?

Click here to view code image

```
public class Box<T> {
  private T item;                                        // (1)

  private static T[] storage = new T[100];               // (2)

  public Box(T item) { this.item = item; }               // (3)

  public T getItem() { return item; }                    // (4)

  public void setItem(T newItem) { item = newItem; }     // (5)

  public static void getAllItems(T newItem) {            // (6)
    T temp;                                              // (7)
  }
}
```

Select the three correct answers.

a. The occurrence of the type parameter `T` at (1)

b. The occurrence of the type parameter `T` at (2)

c. The occurrence of the type parameter `T` at (3)

d. The occurrence of the type parameter `T` at (4)

e. The occurrence of the type parameter `T` at (5)

f. The occurrence of the type parameter `T` at (6)

g. The occurrence of the type parameter `T` at (7)

11.5 Which of the following declarations will not result in compile-time errors? Select the four correct answers.

a. Click here to view code image

```
Map<Integer, Map<Integer, String>> map1
              = new HashMap<Integer, HashMap<Integer, String>>();
```

b. Click here to view code image

```
Map<Integer, HashMap<Integer, String>> map2 = new HashMap<>();
```

c. Click here to view code image

```
Map<Integer, Integer> map3 = new HashMap<Integer, Integer>();
```

d. Click here to view code image

```
Map<? super Integer, ? super Integer> map4
              = new HashMap<? super Integer, ? super Integer>();
```

e. Click here to view code image

```
Map<? super Integer, ? super Integer> map5 = new HashMap<Number, Number
```

f. Click here to view code image

```
Map<? extends Number, ? extends Number> map6
              = new HashMap<Number, Number>();
```

g. Click here to view code image

```
Map <?,?> map7 = new HashMap<?,?>();
```

11.6 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
class Fruit {}
class Apple extends Fruit {}

public class RQ100_15 {
  public static void main(String[] args) {
    List<? extends Apple> lst1 = new ArrayList<Fruit>(); // (1)
    List<? extends Fruit> lst2 = new ArrayList<Apple>(); // (2)
    List<? super Apple> lst3 = new ArrayList<Fruit>();   // (3)
    List<? super Fruit> lst4 = new ArrayList<Apple>();   // (4)
    List<?> lst5 = lst1;                                 // (5)
    List<?> lst6 = lst3;                                 // (6)
    List lst7 = lst6;                                    // (7)
    List<?> lst8 = lst7;                                 // (8)
  }
}
```

Select the one correct answer.

a. (1) will compile, but (2) will not.
b. (3) will compile, but (4) will not.
c. (5) will compile, but (6) will not.
d. (7) will compile, but (8) will not.
e. None of the above

11.7 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_11 {
  public static void main(String[] args) {
    Set set = new TreeSet<String>();
```

```
      set.add("one");
      set.add(2);
      set.add("three");
      System.out.println(set);
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile with unchecked warnings, and will print the elements in the set.

c. The program will compile without unchecked warnings, and will print the elements in the set.

d. The program will compile with unchecked warnings, and will throw an exception at runtime.

e. The program will compile without unchecked warnings, and will throw an exception at runtime.

11.8 Which of the following statements are true about the following program?

Click here to view code image

```
class Vehicle {}
class Car extends Vehicle {}
class Sedan extends Car {}

class Garage<V> {
  private V v;
  public V get() { return this.v; }
  public void put(V v) { this.v = v; }
}

public class GarageAdmin {

  private Object object = new Object();
  private Vehicle vehicle = new Vehicle();
  private Car car = new Car();
  private Sedan sedan = new Sedan();

  public void doA(Garage g) {
    g.put(object);              // (1)
    g.put(vehicle);             // (2)
    g.put(car);                 // (3)
    g.put(sedan);               // (4)
    object  = g.get();          // (5)
    vehicle = g.get();          // (6)
    car     = g.get();          // (7)
    sedan   = g.get();          // (8)
  }
}
```

Select the two correct answers.

a. The call to the `put()` method in statements (1) through (4) will compile.

b. The assignment statement (5) will compile.

c. The assignment statements (6), (7), and (8) will compile.

11.9 Which method calls can be inserted individually at (1) so that the program will compile without warnings?

Click here to view code image

```
import java.util.*;
public class GenParam {
  public static void main(String[] args) {
    List<Number> numList = new ArrayList<>();
    List<Integer> intList = new ArrayList<>();
    // (1) INSERT CODE HERE
  }

  static <T> void move(List<? extends T> lst1, List<? super T> lst2) {}
}
```

Select the three correct answers.

a. `GenParam.move(numList, intList);`

b. `GenParam.<Number>move(numList, intList);`

c. `GenParam.<Integer>move(numList, intList);`

d. `GenParam.move(intList, numList);`

e. `GenParam.<Number>move(intList, numList);`

f. `GenParam.<Integer>move(intList, numList);`

11.10 Given the class below, which declaration statement is not valid?

Click here to view code image

```
class Box<T> {
      Box()         {System.out.println(this);}               // (1)
  <V> Box(V v)      {System.out.println(v);}                  // (2)
  <V> Box(T t, V v) {System.out.println(t + ", " + v);}       // (3)
}
```

Select the one correct answer.

a. `Box<String> ref1 = new Box<>();`

b. `Box<String> ref2 = new Box<>("one");`

c. `Box<String> ref3 = new Box<>(2020);`

d. `Box<Integer> ref4 = new Box<>(2020, "one");`

e. `Box<String> ref5 = new Box<>("one", 2020);`

f. `Box<Integer> ref6 = new Box<>("one", 2020);`

11.9 Wildcard Capture

As we have seen, a wildcard can represent a family of types. However, the compiler needs to have a more concrete notion of a type than a wildcard in order to do the necessary type checking. Internally, the compiler represents the wildcard by some anonymous but specific type. Although this type is unknown, it belongs to the family of types represented by the wildcard. This specific but unknown type is called the *capture of* the wildcard.

Compiler messages about erroneous usage of wildcards often refer to the capture of a wildcard. Here are some examples of such error messages, based on compiling the following code:

Click here to view code image

```
// File: WildcardCapture.java
...
Node<?>                 anyNode;
Node<? super Number>    supNumNode;

Node<Integer> intNode = anyNode;                // (1) Compile-time error!
Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
anyNode.setData("Trash");                       // (3) Compile-time error!
```

The assignment at (1) results in the following rather cryptic error message:

Click here to view code image

```
WildcardCapture.java:10: error: incompatible types: Node<CAP#1> cannot be
converted to Node<Integer>
    Node<Integer> intNode = anyNode;                // (1) Compile-time error!
                            ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

The type of the reference `anyNode` is `Node<CAP#1>`. The name `CAP#1` is used by the compiler to designate the *type capture* of the wildcard (`"capture of ?"`) at (1). The type of the reference `intNode` is `Node<Integer>`. The reference value of a `Node<CAP#1>` cannot be assigned to a `Node<Integer>` reference. Whatever the type capture of the wildcard is, it cannot be guaranteed to be `Integer`, and the assignment is rejected. To put it another way, the assignment involves a narrowing reference conversion, requiring an explicit cast which is not provided: `Node<?>` is the supertype of all invocations of the generic class `Node<E>`.

The error message below for the assignment at (2) shows the type capture `CAP#1` of the lower bounded wildcard at (2) to be `"capture of ? super Number"`. Figure 11.5, p. 583, also shows that the `Node<capture of ? super Number>` and `Node<? extends Number>` types are unrelated.

Click here to view code image

```
WildcardCapture.java:11: error: incompatible types: Node<CAP#1> cannot be
converted to Node<? extends Number>
    Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
                                        ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object super: Number from capture of ? super Number
```

The method call at (3) results in the following error message:

Click here to view code image

```
WildcardCapture.java:12: error: incompatible types: String cannot be converted
CAP#1
    anyNode.setData("Trash");                       // (3) Compile-time error!
                    ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

The type of the reference `anyNode` is `Node<?>` and the type of the formal parameter in the method declaration is `CAP#1`, where `CAP#1` is `"capture of ?"`. The type of the actual parameter in the method call is `String`, which is not compatible with `CAP#1`. The call is not allowed. As we have seen earlier, with a `<?>` reference we cannot put anything into a data structure, except `null`s.

If we have the following method in the class `MyStack`:

Click here to view code image

```
public static <T> void move(MyStack<? extends T> srcStack,
                            MyStack<? super T> dstStack) {
  while (!srcStack.isEmpty())
    dstStack.push(srcStack.pop());
}
```

and we try to compile the following client code in the source file `MyStackUser.java`:

Click here to view code image

```
MyStack<?> anyStack;
MyStack.move(anyStack, anyStack);   // Compile-time error!
```

the compiler issues the following error message:

Click here to view code image

```
MyStackUser.java:68: error: method move in class MyStack<E#2> cannot be applie
given types;
      MyStack.move(anyStack, anyStack);     // Compile-time error!
             ^
  required: MyStack<? extends E#1>,MyStack<? super E#1>
  found: MyStack<CAP#1>,MyStack<CAP#2>
  reason: cannot infer type-variable(s) E#1
    (argument mismatch; MyStack<CAP#2> cannot be converted to MyStack<? super
  where E#1,E#2 are type-variables:
    E#1 extends Object declared in method <E#1>move(MyStack<? extends E#1>,My-
Stack<? super E#1>)
    E#2 extends Object declared in class MyStack
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Object from capture of ?
```

```
      CAP#2 extends Object from capture of ?
```

The error message shows that each occurrence of a wildcard in a statement is represented by a distinct type capture, namely `CAP#1` and `CAP#2`. We see that the signature of the `move()` method is `move(MyStack<? extends E#1>, MyStack<? super E#1>)`. The type of the reference `anyStack` is `MyStack<?>`. The static types of the two arguments in the method call are `MyStack<CAP#1>` and `MyStack<CAP#2>`, where `CAP#1` and `CAP#1` designate two distinct type captures (`capture of ?`). The signature of the argument list is (`MyStack<CAP#1>, MyStack<CAP#2>`). The type parameter `T` cannot be inferred from the types of the arguments, as the stacks are considered to be of two different types, and therefore, the call is rejected.

Capture Conversion

Consider the following non-generic method which does not compile:

Click here to view code image

```
    static void fillWithFirstV1(List<?> list) {
      Object firstElement = list.get(0);        // (1)
      for (int i = 1; i < list.size(); i++)
        list.set(i, firstElement);              // (2) Compile-time error
    }
```

The method should fill any list passed as an argument with the element in its first position. The call to the `set()` method at (2) is not permitted, as a *set* operation is not possible with a `<?>` reference (see Table 11.3, p. 590). Using the unbounded wildcard `?` to parameterize the list does not work. We can replace the wildcard with a type parameter of a *generic method*, as follows:

Click here to view code image

```
  static <E> void fillWithFirstV2(List<E> list) {
      E firstElement = list.get(0);             // (3)
      for (int i = 1; i < list.size(); i++)
        list.set(i, firstElement);              // (4)
    }
```

Since the type of the argument is `List<E>`, we can set and get objects of type `E` from the list. We have also changed the type of the reference `firstElement` from `Object` to `E` in order to set the first element in the list.

It turns out that if the first method `fillWithFirstV1()` is reimplemented with a call to the generic method `fillWithFirstV2()`, it all works well:

Click here to view code image

```
static void fillWithFirstV3(List<?> list) {
  fillWithFirstV2(list);                   // (5) Type conversion
}
```

The wildcard in the argument of the `fillWithFirstV3()` method has a type capture. In the call to the `fillWithFirstV2()` method at (5), this type capture is converted to the type `E`. This conversion is called *capture conversion*, and it comes into play under certain conditions, which are beyond the scope of this book.

11.10 Flexibility with Wildcard Parameterized Types

Nested Wildcards

In this subsection, the examples make use of type parameters that are specified for the interfaces `Collection<E>`, `Set<E>`, and `Map<K,V>` in the `java.util` package (Chapter 15, p. 781). In the discussion below, the important fact to keep in mind is that the interface `Set<E>` is a subinterface of `Collection<E>`.

We have seen that the subtype relationship is invariant for the unbounded type parameter `<T>`:

Click here to view code image

```
Collection<Number> colNum;
Set<Number> setNum;
Set<Integer> setInt;
colNum = setNum; // (1) Set<Number> <: Collection<Number>
colNum = setInt; // (2) Compile-time error!
```

The same is true when concrete parameterized types are used as actual type parameters, implementing what are called *nested parameterized types*—that is, using parameterized types as type parameters.

Click here to view code image

```
Collection<Collection<Number>> colColNum; // Collection of Collections of Numb
Set<Collection<Number>>        setColNum; // Set of Collections of Number
Set<Set<Integer>>              setSetInt; // Set of Sets of Integer
colColNum = setColNum;                    // (3) Set<Collection<Number>> <:
                                          //     Collection<Collection<Number>
colColNum = setSetInt;                    // (4) Compile-time error!
```

```
setColNum = setSetInt;                     // (5) Compile-time error!
```

Again, we can use the upper bounded wildcard to induce subtype covariance. The upper bounded wildcard is applied at the *top level* in the code below. The assignment below at (8) is not compatible because `Set<Set<Integer>>` is *not* a subtype of

Click here to view code image

```
Collection<? extends Collection<Number>> colExtColNum;
colExtColNum = colColNum;         // (6) Collection<Collection<Number>> <:
                                  //     Collection<? extends Collection<Number>
colExtColNum = setColNum;         // (7) Set<Collection<Number>> <:
                                  //     Collection<? extends Collection<Number>
colExtColNum = setSetInt;         // (8) Compile-time error!
```

In the code below, the wildcard is applied at the *innermost* level:

Click here to view code image

```
Collection<Collection<? extends Number>> colColExtNum;
colColExtNum = colColNum;         // (9)  Compile-time error!
colColExtNum = setColNum;         // (10) Compile-time error!
colColExtNum = setSetInt;         // (11) Compile-time error!
```

The assignments above show that the upper bounded wildcard induces subtype covariance *only* at the top level. At (9), type `A` (=`Collection<Number>`) is a subtype of type `B` (=`Collection<? extends Number>`), but because a subtype covariance relationship does not hold between parameterized types, the type `Collection<A>` (=`Collection <Collection<Number>>`) is *not* a subtype of `Collection<B>` `(= Collection<Collection<? extends Number>>).`

The above discussion also applies when a parameterized type has more than one type parameter:

Click here to view code image

```
Map<Number, String>  mapNumStr;
Map<Integer, String> mapIntStr;
mapNumStr = mapIntStr;              // (12) Compile-time error!
```

Again, the upper bounded wildcard can only be used at the top level to induce subtype covariance:

Click here to view code image

```
Map<Integer, ? extends Collection<String>> mapIntExtColStr;
Map<Integer, Collection<? extends String>> mapIntColExtStr;
Map<Integer, Collection<String>>           mapIntColStr;
Map<Integer, Set<String>>                  mapIntSetStr;
mapIntExtColStr = mapIntColStr;// (13) Map<Integer, Collection<String>> <:
```

```
                               //      Map<Integer, ? extends Collection<Strin
  mapIntExtColStr = mapIntSetStr;// (14) Map<Integer, Set<String>> <:
                               //      Map<Integer, ? extends Collection<Strin
  mapIntColStr    = mapIntSetStr;    // (15) Compile-time error!
  mapIntColExtStr = mapIntColStr;    // (16) Compile-time error!
  mapIntColExtStr = mapIntSetStr;    // (17) Compile-time error!
```

Wildcard Parameterized Types as Formal Parameters

We now examine the implications of using wildcard parameterized types to declare formal parameters of a method.

We want to add a method in the class `MyStack<E>` (Example 11.10, p. 598) for moving the elements of a source stack to the current stack. Here are three attempts at implementing such a method for the class `MyStack<E>`:

Click here to view code image

```
  public void moveFromV1(MyStack<E> srcStack) {            // (1)
    while (!srcStack.isEmpty())
      this.push(srcStack.pop());
  }
  public void moveFromV2(MyStack<? extends E> srcStack) {  // (2)
    while (!srcStack.isEmpty())
      this.push(srcStack.pop());
  }
  public void moveFromV3(MyStack<? super E> srcStack) {    // (3)
    while (!srcStack.isEmpty())
      this.push(srcStack.pop());                           // Compile-time error
  }
```

Given the following three stacks:

Click here to view code image

```
  MyStack<Number> numStack   = new MyStack<>();                  // Stack of Num
  numStack.push(5.5); numStack.push(10.5); numStack.push(20.5);
  MyStack<Integer> intStack1 = new MyStack<>();                  // Stack of Int
  intStack1.push(5); intStack1.push(10); intStack1.push(20);
  MyStack<Integer> intStack2 = new MyStack<>();                  // Stack of Int
  intStack2.push(15); intStack2.push(25); intStack2.push(35);
```

We can only move elements between stacks of the same concrete type with the method at (1). The compile-time error below is due to the fact that `MyStack<Integer>` is not a subtype of `MyStack<Number>`.

Click here to view code image

```
  intStack1.moveFromV1(intStack2); // Ok.
  numStack.moveFromV1(intStack2);  // Compile-time error!
```

We can also move elements from a stack of type `MyStack<? extends E>` to the current stack, using the method at (2). This is possible because a reference of a type `MyStack<? extends E>` can refer to a stack with objects of type `E` or its subclass, and the *get* operation (i.e., the `pop()` method) is permissible, returning an object which has an actual type bounded by the upper bound `E`. The returned object can always be put into a stack of type `E` or its supertype.

Click here to view code image

```
intStack1.moveFromV2(intStack2);  // Pop from intStack2. Push on intStack1.
numStack.moveFromV2(intStack2);   // Pop from intStack2. Push on numStack.
```

The method at (3) will only allow `Object`s to be popped from a stack of type `MyStack<? super E>`, which could only be pushed onto a stack of type `Object`. Since `E` cannot be determined at compile time, the `push()` operation on the current stack is not permitted. Of the first two methods, the method at (2) is more flexible in permitting the widest range of calls for copying from the source stack to the current stack.

Similarly, we can add a method in the class `MyStack<E>` for moving the elements of the current stack to a destination stack:

Click here to view code image

```
public void moveToV1(MyStack<E> dstStack) {                  // (4)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}

public void moveToV2(MyStack<? extends E> dstStack) {        // (5)
  while (!this.isEmpty())
    dstStack.push(this.pop());                               // Compile-time erro
}

public void moveToV3(MyStack<? super E> dstStack) {          // (6)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}
```

In the method at (5), the reference of type `MyStack<? extends E>` does not allow any *set* operations (in this case, the `push()` method) on the destination stack. The method at (6) provides the most flexible solution, as a reference of type `MyStack<? super E>` permits *set* operations for objects of type `E` or its subtypes:

Click here to view code image

```
intStack1.moveToV1(intStack2);    // Pop from intStack1. Push on intStack2.
intStack1.moveToV1(numStack);     // Compile-time error!

intStack1.moveToV3(intStack2);    // Pop from intStack1. Push on intStack2.
```

```
intStack1.moveToV3(numStack);     // Pop from intStack1. Push on numStack.
```

Evidently, the method at (6) is more flexible in permitting the widest range of calls for copying from the current stack to the destination stack.

Based on the discussion above, we can write a *generic method* for moving elements from a source stack to a destination stack. The following method signature is preferable, where objects of type `E` or its subtypes can be popped from the source stack and pushed onto a destination stack of type `E` or its supertype:

Click here to view code image

```
public static <E> void move(MyStack<? extends E> srcStack,  // (7)
                            MyStack<? super E> dstStack) {
  while (!srcStack.isEmpty())
    dstStack.push(srcStack.pop());
}

// Client code
MyStack.move(intStack1, intStack2);
MyStack.move(intStack1, numStack);
MyStack.move(numStack, intStack2);     // Compile-time error!
```

It is a common idiom to use wildcards as shown above in the method at (7), as the upper bounded wildcard (`? extends Type`) can be used to *get* objects from a data structure, and the lower bounded wildcard (`? super Type`) can be used to *set* objects in a data structure. Using wildcards in the method signature can increase the utility of a method, especially when explicit type parameters are specified in the method call.

Recursive Type Bounds Revisited

The class `MonoNode` and the interface `IMonoLink<E>` are declared in Example 11.3, p. 572, and the class `BiNode` and the interface `IBilink<E>` are declared in Example 11.5, p. 574. See also Figure 11.1, p. 574.

Click here to view code image

```
class MonoNode<E> implements IMonoLink<E> {
  private E              data;    // Data
  private IMonoLink<E>   next;    // Reference to next node       (1)
  // ...
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E> previous;      // Reference to previous node   (2)
  // ...
}
```

Note that the `next` field has the type `IMonoLink<E>`, but the `previous` field has the type `IBiLink<E>`, and that the type `IBiLink<E>` is a subtype of the type `IMonoLink<E>`. This means that when iterating over a linked structure constructed from nodes that implement the `IBiLink<E>` interface, we have to be careful. The method `traverseBinTree()` below traverses a binary tree constructed from such nodes. The method prints the data in the nodes. Note that it is necessary to cast the reference value returned by the `getNext()` method, as shown at (3), from the supertype `IMonoLink<E>` to the subtype `IBiLink<E>`.

Click here to view code image

```
public static <E> void traverseBinTree(IBiLink<E> root) {
  if (root.getPrevious() != null)
    traverseBinTree(root.getPrevious());
  System.out.print(root.getData() + ", ");
  if (root.getNext() != null)
    traverseBinTree((IBiLink<E>)root.getNext());  // (3) Cast necessary.
}
```

Example 11.11 declares a class called `RecNode`. The header of this class at (1) is declared as follows:

Click here to view code image

```
abstract class RecNode<E, T extends RecNode<E, T>>
```

The class specifies two type parameters: `E` and `T`. The type parameter `E` stands for the type of the data in the node, (2). The type parameter `T` stands for the type of the `next` field in the node, (3). It has the upper bound `RecNode<E, T>`, meaning that `T` must be a subtype of the bound—that is, of `RecNode<E, T>`. In other words, the class `RecNode` can only be parameterized by itself or its subclasses. The two parameters `E` and `T` are used in the class for their respective purposes.

Example 11.11 *Using Recursive Bounds*

Click here to view code image

```
abstract class RecNode<E, T extends RecNode<E, T>> {       // (1)
  private E data;                                          // (2)
  private T next;                                          // (3)

  RecNode(E data, T next) {
    this.data = data;
    this.next = next;
  }
  public void setData(E obj)  { data = obj; }
  public E    getData()       { return data; }
  public void setNext(T next) { this.next = next; }
  public T    getNext()       { return next; }
  @Override public String toString() {
    return this.data + (this.next == null ? "" : ", " + this.next);
  }
}
```

Click here to view code image

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>> {          // (4)

  private RecBiNode<E>  previous;     // Reference to previous node   // (5)

  RecBiNode(E data, RecBiNode<E> next, RecBiNode<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  public void setPrevious(RecBiNode<E> previous) { this.previous = previous; }
  public RecBiNode<E> getPrevious()              { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
        this.getData() + (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

Example 11.11 declares another class, called `RecBiNode`. The header of this class at (4) is declared as follows:

Click here to view code image

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>>
```

Note that the class has only one type parameter, `E`, that represents the data type in the node. The class extends the `RecNode` class, which is parameterized with the data type `E` and the type `RecBiNode<E>` to represent the type of the `next` field in the superclass `RecNode`. The class `RecBiNode` also declares a `previous` field of type `RecBiNode<E>` at (5), and the corresponding getter and setter methods. The upshot of this class declaration is that, for a node of type `RecBiNode<E>`, both the `next` and the `previous` fields have the same type as a node of this class. The traversal method can now be written without using any casts, passing it a node of the subtype `RecBiNode<E>`:

Click here to view code image

```
  public static <E> void traverseBinTree(RecBiNode<E> root) {     // (2)
    if (root.getPrevious() != null)
      traverseBinTree(root.getPrevious());
    System.out.print(root.getData() + ", ");
    if (root.getNext() != null)
      traverseBinTree(root.getNext());             // No cast necessary!
  }
```

The class declaration at (1) in Example 11.11 uses what is called a *recursive type bound* in its constraint `T extends RecNode<E,T>`. New subtypes of the class `RecNode` can be implemented using this idiom, and the type of the `next` field will be the same as the subtype, as in the case of the subtype `RecBiNode<E>`. Recursive type bounds allow subclasses to use their own type, avoiding explicit casts to convert from the superclass.

Earlier in this chapter we saw an example of a recursive type bound defined by the constraint `T extends Comparable<T>`. Another example of a recursive type bound is the declaration of the `Enum<E extends Enum<E>>` class in the Java standard library.

11.11 Type Erasure

Understanding translation by type erasure aids in understanding the restrictions and limitations that arise when using generics in Java. Although the compiler generates generic-free bytecode, we can view the process as a source-to-source translation that generates non-generic code from generic code.

The translated code has no information about type parameter. That is, the type parameters have been *erased*—hence the term *type erasure*. This involves replacing the usage of the type parameters with concrete types, and inserting suitable type conversions to ensure type correctness. In certain situations, *bridge methods* are also inserted for backward compatibility.

Translation by Type Erasure

The process of determining the *erasure* of a type—that is, what a type in the source code should be replaced with—uses the following rules:

- Drop all type parameter specifications from parameterized types.
- Replace any type parameter as follows:
    a. Replace it with the erasure of its bound, if it has one.
    b. Replace it with `Object`, if it has none.
    c. Replace it with the erasure of the *first* bound, if it has multiple bounds.

Table 11.4 shows examples of translation by erasure for some representative types, and the rules that are applied.

Table 11.4 *Examples of Type Erasure*

Type	Erasure	Rule no.

`List<E>`
`List<Integer>`
`List<String>`
`List<List<String>>`
`List<? super Integer>`
`List<? extends Number>`
	`List`	1

`List<Integer>[]`	`List[]`	1
`List`	`List`	1
`int`	`int`	For any primitive type
`Integer`	`Integer`	For any non-generic type
`class Subclass extends Superclass implements Comparable<Subclass> {...}`	`class Subclass extends Superclass implements Comparable {...}`	1
`public static <T extends Comparable<? super T>> T max(T obj1, T obj2) { ... }`	`public static Comparable max(Comparable obj1, Comparable obj2) { ... }`	2a. The first bound is `Comparable`.
`public static <T> T doIt(T t) { T lv = t; }`	`public static Object doIt(Object t) { Object lv = t; }`	2b
`T extends MyClass & Comparable<T> & Serializable`	`MyClass`	2c. The first bound is `MyClass`.

The following code mixes legacy and generic code. Note that a `ClassCastException` is expected at (5) because the type-safety of the stack of `String` has been compromised.

Click here to view code image

```
// Pre-erasure code
List<String> strList = new ArrayList<>(); // (0)
List list = strList;        // (1) Assignment to non-generic reference is ok.
strList = list;             // (2) warning: unchecked conversion
strList.add("aha");         // (3) Method call type-safe.
list.add(23);               // (4) warning: [unchecked] unchecked call to add(E
                            //     as a member of the raw type java.util.List
System.out.println(strList.get(1).length());  // (5) ClassCastException
```

It is instructive to compare the corresponding lines of code in the pre-erasure code above and the post-erasure results shown below. A cast is inserted to convert from `Object` type to `String` type at (5'). This is necessary because post-erasure code can only get an `Object` from the list, and in order to call the `length()` method, the reference value of this object must be converted to `String`. It is this cast that is the cause of the exception at runtime.

Click here to view code image

```
// Post-erasure code
List strList = new ArrayList();                         // (0')
List list = strList;                                    // (1')
strList = list;                                         // (2')
strList.add("aha");                                     // (3')
list.add(Integer.valueOf(23));                          // (4')
System.out.println(((String)strList.get(1)).length()); // (5') Cast inserted.
```

Bridge Methods

Bridge methods are inserted in subclasses by the compiler to ensure that overriding of methods works correctly. The canonical example is the implementation of the `Comparable` interface. The post-erasure code of the class `CmpNode<E>` from Example 11.8 is shown below. A second `compareTo()` method has been inserted by the compiler at (2), whose method signature is `compareTo(Object)`. This is necessary because, without this method, the class would not implement the `Comparable` interface, as the post-erasure `compareTo(Object)` method of the interface would not be overridden correctly in the erasure code.

Click here to view code image

```
class CmpNode extends Node implements Comparable {
  CmpNode(Object data, CmpNode next) {
    super(data, next);
  }
  public int compareTo(CmpNode node2) {        // (1) Implemented method erasur
    return this.compareTo(node2.getData());
  }
  public int compareTo(Object node2) {         // (2) Inserted bridge method.
    return this.compareTo((CmpNode)node2);     // Calls the method at (1).
  }
}
```

11.12 Implications for Overloading and Overriding

Before discussing the implications generics have for overloading and overriding, we need a few definitions regarding method signatures.

Method Signatures Revisited

Method signatures play a crucial role in overloading and overriding of methods. The method signature comprises the method name and the formal parameter list. Two methods (or constructors) have the *same signature* if both of the following conditions are fulfilled:

- They have the *same name.*
- They have the *same formal parameter types.*

Two methods (or constructors) have the *same formal parameter types* if both of the following conditions are fulfilled:

- They have the *same number* of *formal parameters* and *type parameters*.
- The formal parameters and the bounds of the type parameters are the same after the occurrences of formal parameters in one are substituted with the corresponding types from the second.

The signature of a method `m()` is a *subsignature* of the signature of another method `n()`, if either one of these two conditions hold:

- Method `n()` has the *same* signature as method `m()`, or
- The signature of method `m()` is the same as the *erasure* of the signature of method `n()`.

The signatures of the two methods `m()` and `n()` are *override-equivalent* if either one of these two conditions hold:

- The signature of method `m()` is a *subsignature* of the signature of method `n()`, or
- The signature of method `n()` is a *subsignature* of the signature of method `m()`.

Implications for Overloading

Given the definitions above, we can now state that two methods are *overloaded* if they have the same name, but their signatures are *not override-equivalent.* Given the following three generic method declarations in a class:

Click here to view code image

```
static <T> void merge(MyStack<T> s1, MyStack<T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? super T> s2) { /*...*/ }
```

After erasure, the signature of all three methods is:

```
merge(MyStack, MyStack)
```

That is, the signatures of the methods are override-equivalent, hence these methods are *not* overloaded. A class cannot contain two methods with override-equivalent signatures, and the compiler will report an error.

These three methods:

Click here to view code image

```
static <T> void merge(Node<T> s1, MyStack<T> s2)                { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, Node<? super T> s2)       { /*...*/ }
```

have the following signatures after erasure, respectively:

```
merge(Node, MyStack)
merge(MyStack, MyStack)
merge(MyStack, Node)
```

We can see that no two signatures are override-equivalent. Therefore, the three methods are overloaded.

The declaration of the class `MethodSGN` below shows some variations on the method signature. The resulting signature of the method header (which includes the method signature) is shown after erasure in the comment corresponding to the method.

Click here to view code image

```
class MethodSGN<T> {
  void doIt(boolean b) {}                          // (1) void doIt(boolean)

  void doIt(T t) {}                                // (2) void doIt(Object)

  List<StringBuilder> doIt(StringBuilder sb) {     // (3) List doIt(StringBuilde
    return null;
  }

  <E extends Comparable<E>> void doIt(E element) // (4) void doIt(Comparable)
    {}
  <E> E doIt(MyStack<? extends E> stack) {         // (5) Object doIt(MyStack)
    return null;
  }
}
```

Adding any of the method declarations given below to the class `MethodSGN` would be an error, as each one of these method declarations has a method signature that is the same as one of the methods already in the class—that is, the signatures are override-equivalent.

Click here to view code image

```
  void doIt(Object obj) {}                         // (2') void doIt(Object)

  <E extends StringBuilder> List<E> doIt(E sb) {   // (3') List doIt(StringBuild
    return null;
  }

  void doIt(Comparable<T> element) {}              // (4') void doIt(Comparable)

  <E> E doIt(MyStack<? super E> stack) {           // (5') Object doIt(MyStack)
    return null;
  }
```

Implications for Overriding

The following conditions (referred to as *override criteria*) should be satisfied in order for a subtype method to override a supertype method:

- The signature of the subtype method is a *subsignature* of the signature of the supertype method (which is discussed in this subsection).
- Their *return types* should be compatible.
- Their `throws` *clauses* should be compatible (§7.5, p. 388).

Here we discuss the implication of method signatures for overriding.

The `@Override` *Annotation*

We can solicit the aid of the compiler to ensure that a method declaration in a subtype correctly overrides a method from its supertype. If a method declaration is preceded by the annotation `@Override` (§25.5, p. 1578), the compiler will issue an error if the method in the subtype does not override a method from its supertype. The examples in this book make heavy use of this annotation.

Example 11.12 illustrates the use of this annotation. The intention in the class `CmpNode` is to override the `equals()` method from the `Object` class and the `compareTo()` method from the `Comparable` interface. The error messages alert us to the fact that the annotated methods do not override any methods. The method signatures are not subsignatures of any methods that are inherited. The formal parameters are not correct for overriding at (1) and (2). Correct method headers are shown at (1') and (2').

Example 11.12 *Using the* `@Override` *Annotation*

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override
  public boolean equals(CmpNode node2) {             // (1) Compile-time error.
//public boolean equals(Object node2) {              // (1') Correct header.
    return this.compareTo(node2) == 0;
  }

  @Override
  public int compareTo(Object node2) {                // (2) Compile-time error.
//public int compareTo(CmpNode<E> node2) {            // (2') Correct header
    return this.getData().compareTo(node2.getData());
  }
}
```

Compiling the class `CmpNode`:

Click here to view code image

```
>javac CmpNode.java
...
CmpNode.java:8: method does not override or implement a method from a supertyp
  @Override
  ^
...
CmpNode.java:14: method does not override or implement a method from a superty
  @Override
  ^
```

Overriding Methods from Non-Generic Supertype

In Example 11.13, the signature at (1') is the same as the signature at (1): `set(Integer)`. The signature at (2') is the same as the *erasure* of the signature at (2): `set(List)`. The method at (2') shows a non-generic subtype method overriding a supertype method that uses generics. This is needed for legacy code: *Legacy supertypes* can be generified without it having consequences for any subtypes, as the signature of a subtype method that overrides a supertype method will be the same as the erasure of the signature of this supertype method.

Example 11.13 *Subsignatures*

Click here to view code image

```
import java.util.List;

class SupA {
  public void set(Integer ref)        {/*...*/}    // (1)
  public void set(List<Integer> list) {/*...*/}    // (2)
}

class SubA extends SupA {
  @Override public void set(Integer ref) {/*...*/}  // (1') same as at (1)
  @Override public void set(List list) {/*...*/}// (2') same as the erasure at
}
```

Overriding Methods from Generic Supertype

In Example 11.14, both the subclasses `SubB1` and `SubB2` are subtypes of the parameterized supertype `SupB<Number>`—that is, `T` is `Number` in `SupB<T>`. At compile time, the signatures of the methods in `SubB1` are the same as the signatures of the methods in `SupB`; therefore, the methods are overridden. After erasure, the methods in `SupB` are equivalent to:

Click here to view code image

```
public void set(Object t) {/*...*/}                  // (1)
public Object get()       {return null;}             // (2)
```

The compiler adds the following bridge method in `SubB1` in order for overriding to work properly at runtime:

Click here to view code image

```
public void set(Object t) {set((Number)t);}        // (1')
```

It does not add a bridge method for the `get()` method in `SubB1` because of covariant return: The return type `Number` for the method `get()` in `SubB1` is a subtype of the return type `Object` of the method `get()` in `SupB`.

Example 11.14 *Overriding from Generic Supertype*

Click here to view code image

```
class SupB<T> {
  public void set(T t) {/*...*/}                      // (1)
  public T get()       {return null;}                 // (2)
}

class SubB1 extends SupB<Number> {
  @Override public void set(Number num) {/*...*/} // (1a) Overrides
  @Override public Number get()        {return 0;} // (2a) Overrides
}

class SubB2 extends SupB<Number> {
  @Override public void set(Object obj) {/*...*/} // (1b) Error: same erasure
  @Override public void set(Long l)     {/*...*/} // (1c) Error: overloads

  @Override public Object get() {              // (2b) Error: incompatible return
    return null;
  }
}
```

We now examine the methods in `SubB2`. The `set()` method at (1b) has the *same signature* as *the erasure* of the signature of the `set()` method at (1) in the supertype `SupB`. If overriding were allowed, the bridge method added would result in *two* methods with the same signature `set(Object)` in `SubB2`. Two methods with the same signature are not permitted in the same class—called a *name clash*—therefore, (1b) is not allowed.

The method `set()` at (1c) is overloaded because its signature is different from the other `set()` methods in `SubB2` and `SupB`. It would compile if the `@Override` annotation was removed. The method `get()` at (2b) has the return type `Object`, while the `get()` method in `SupB<Number>` has the return type `Number`. The return types are not covariant, and (2b) is rejected.

Example 11.15 shows a typical error where a generic supertype is extended, but its parameterization is missing in the `extends` clause of the subtype, as shown at (2). The `set()` method in `SubZ` neither overrides nor overloads the method at (1). Both methods have the same signature after erasure: `set(Object)`. Adding a bridge method in `SubZ` to make the overriding work properly would result in a name clash. (1a) is rejected.

Example 11.15 *Missing Supertype Parameterization*

Click here to view code image

```
class SupZ<T> {
  public void set(T t) {/*...*/}               // (1)
}

class SubZ<E> extends SupZ {                   // Superclass not parameterized
  @Override public void set(E e) {/*...*/}     // (1a) Error: same erasure
}
```

Genericity and Inherited Methods

The subsignature requirement for overriding means that the signature of the subtype method must be the *same* as that of the supertype method, or it must be the same as the *erasure* of the signature of the supertype method. Note the implication of the last sentence: The signature of the subtype method must be the same as the *erasure of the supertype method, not* the other way around. The converse is neither overloading nor overriding, but a *name clash* and is reported as an error.

The subsignature requirement also implies that a *generic subtype method cannot override a non-generic supertype method*. In other words, genericity cannot be added to an inherited method. This case is illustrated in Example 11.16. It is the erasures of the signatures of the generic methods in the subtype that are the same as the signatures of the non-generic methods in the supertype. Overriding requires the converse. A name clash is generally the reason why neither overriding nor overloading is permitted.

Example 11.16 *Genericity Cannot be Added to Inherited Methods*

Click here to view code image

```
class SupJ {
  public void set(Object obj) {/*...*/}           // (1)
  public Object get()    {return null;}           // (2)
}

class SubJ extends SupJ {
  @Override public <T> void set(T t) {/*...*/}    // (1a) Error: same erasure
  @Override public <S> S get()   {return null;}   // (2a) Error: same erasure
}
```

Overriding Abstract Methods from Multiple Superinterfaces

The code below illustrates the simple case where the subinterface `TaskAB` at (3) extends the two superinterfaces, `TaskA` and `TaskB`, whose abstract methods are *override-equivalent*—that is, they have the same signature. The abstract method declared at (4) in the subinterface `TaskAB` *overrides* the two abstract method declarations from the superinterfaces. This method represents the two abstract method declarations from the superinterfaces. The compiler can also infer it from the inherited methods, and its declaration at (4) can be omitted.

Click here to view code image

```
interface TaskA { Object compute(Integer iRef1); }   // (1)
interface TaskB { Object compute(Integer iRef2); }   // (2)
interface TaskAB extends TaskA, TaskB {              // (3)
  @Override Object compute(Integer iRef3);           // (4) Can be omitted.
}

interface TaskC { String compute(Integer iRef4); }   // (5)
interface TaskABC extends TaskA, TaskB, TaskC {      // (6)
  @Override String compute(Integer iRef5);           // (7a) Can be omitted.
//@Override Object compute(Integer iRef6);           // (7b) Compile-time erro
}
```

The subinterface `TaskABC` at (6) extends the superinterfaces `TaskA`, `TaskB`, and `TaskC`. The abstract methods of the three superinterfaces are override-equivalent, since they have the same method signature:

`compute(Integer)`. The abstract method at (5) in `TaskC` can *override* the other two methods with its *covariant* return, as `String` type is a subtype of `Object` type. The compiler infers that the abstract method at (5) can represent the methods from the superinterfaces in the subinterface `TaskABC`. Again we can omit the abstract method declaration at (7a), as the compiler can infer it. A client of course must make sure that it implements the correct abstract method when implementing the interface `TaskABC`.

If the abstract method declaration at (7a) is replaced with the abstract method declaration at (7b) where the return type is `Object`, the code will not compile, as this method cannot override the abstract method declaration at (5) declared with the return type `String`. This is an example of a *name clash*—that is, the inherited methods from the superinterfaces are override-equivalent, but the method at (7b) cannot override them all, thus resulting in a compile-time error.

The next example shows that the multiple abstract methods inherited by the `ContractZ` are not override-equivalent—in fact they are overloaded, and both are inherited from their respective interfaces by the subinterface.

Click here to view code image

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
interface ContractZ extends ContractX, ContractY {
  @Override void doIt(int d);                             // Can be omitted.
  @Override void doIt(double d);                          // Can be omitted.
}
```

So far all examples have involved methods without generics, where a subtype method could override a supertype method that had the same signature.

A subtype method *without generics* can *override* supertype methods *with generics* that have the same signature *after type erasure*. After type erasure of the `bake()` methods at (1) and (2) in the code below, all three `bake()` methods at (1), (2), and (3) are override-equivalent with the signature `bake(List)`. However, it is the `bake()` method at (3) that *overrides* the `bake()` methods at (1) and (2). This `bake()` method *without generics* at (3) logically represents all the inherited `bake()` methods in the `Bakeable` subinterface. The implementation of the `Bakeable` subinterface thus requires implementation of the `bake()` method without generics declared at (3).

Click here to view code image

```
class Ingredient { }

interface PizzaBakeable {
  void bake(List<Ingredient> ingredients);  // (1) After type erasure: bake(Li
}

interface CalzoneBakeable {
  void bake(List<Ingredient> ingredients);  // (2) After type erasure: bake(Li
}

interface SimplyBakeable {
  void bake(List ingredients);              // (3) Signature: bake(List)
}

interface Bakeable extends SimplyBakeable, PizzaBakeable, CalzoneBakeable {
  @Override void bake(List ingredients);    // Can be omitted.
}
```

In the code above, the `bake()` method without generics at (3) that *overrides* the other two inherited `bake()` methods, and thus represents all the inherited `bake()` methods in the subinterface, can be inferred by *type erasure of any of the inherited* `bake()` *methods with generics*, since they are all override-equivalent.

From the examples in this subsection, we see that override-equivalent methods in superinterfaces can be legally inherited by a subinterface, if a method can be inferred that *overrides* all inherited override-equivalent methods (after any type erasure), and which then represents these methods in the subinterface. In the case of override-equivalent methods *with generics* from multiple superinterfaces, the type erasure of *any* of the override-equivalent methods is always a candidate that can override and represent these methods in the subinterface.

11.13 Limitations and Restrictions on Generic Types

In this section we take a look at implications and restrictions on generic types for instance tests, casting, arrays, variable arity parameters, exception handling, nested classes, and enum types.

Reifiable Types

Concrete parameterized types are used by the compiler and then translated by erasure to their raw types, losing information about the parameterization in the process. In other words, only the raw types of these concrete parameterized types are available at runtime. For example, `List<Integer>` and `List<String>` are both erased to the raw type `List`. The same applies to unbounded parameterized types: `List<E>` is erased to `List`.

Non-generic types are not affected by type erasure, and therefore, have *not* lost any information and are, therefore, available fully at runtime. For example, the types `Integer` and `String` remain intact and are present unchanged at runtime.

Types that are completely available at runtime are known as *reifiable types*—that is, type erasure has *not* removed any important information about them (see Table 11.5). Types whose information has been affected by erasure are called *nonreifiable types* (see Table 11.6).

Note that unbounded wildcard parameterized types (`Node<?>`, `MyStack<?>`) are reifiable, whereas concrete parameterized types (`Node<Number>`, `MyStack<String>`) and bounded wildcard parameterized types (`Node<? extends Number>`, `MyStack<? super String>`) are non-reifiable.

As we shall see in the rest of this section, certain operations in Java are only permitted on reifiable types (as their type information is fully intact and available at runtime), and not on non-reifiable types (as their type information is not fully available at runtime, since it has been affected by type erasure).

Table 11.5 *Examples of Reifiable Types*

Reifiable type	Example
A primitive type	`int, double, boolean`
A non-generic type	`Exception, System, Math, Number`
A raw type	`List, ArrayList, Map, HashMap`
A parameterized type in which all type arguments are *unbounded wildcards* (unbounded wildcard parameterized type)	`List<?>, ArrayList<?>, Map<?,?>, HashMap<?,?>`
An *array type* whose component type is reifiable	`double[], Number[], List[], HashMap<?,?>[], Number[][]`

Table 11.6 *Examples of Non-Reifiable Types*

Non-Reifiable type	Example
A type parameter	`E, T, K, V`
A parameterized type with *concrete or unbounded type parameters* (concrete or unbounded parameterized type)	`List<E>, List<String>, ArrayList<Integer>, HashMap<String, Number>`

	`Map<K, V>`

A parameterized type with a *bound* (bounded wildcard parameterized type)	`List<? extends Object>,`
	`ArrayList<? extends Number>,`
	`Comparable<? super Integer>`

An *array type* whose component type is non-reifiable	`List<E>[],`
	`ArrayList<Number>[],`
	`Comparable<? super Integer>[],`
	`HashMap<K, V>[]`

Implications for the `instanceof` operator

Although the discussion here is about the `instanceof` type comparison operator, it applies equally to the `instanceof` pattern match operator.

At (1) below, we want to determine whether the object referenced by `obj` is an instance of the concrete parameterized type `MyStack<Integer>`—that is, whether it is a stack of `Integer`.

Click here to view code image

```
Object obj;
...
boolean isIntStack = obj instanceof MyStack<Integer>; // (1) Compile-time erro
```

The post-erasure code for (1) is equivalent to the following statement:

Click here to view code image

```
boolean isIntStack = obj instanceof MyStack;          // (1')
```

The statement at (1') cannot perform the `instanceof` type comparison as expected at (1), since the type erasure has removed the information about the concrete type parameter `Integer`—that is, the type `MyStack<Integer>` is non-reifiable. The compiler issues an error because the type `Object` cannot be *safely cast* to the type `MyStack` at runtime.

In the following code, the fact that `IStack<Integer>` is a supertype of `MyStack<Integer>` can be checked at compile time—that is, the type `MyStack<Integer>` can be considered a *refinement* of the type `IStack<Integer>`. Such refinement of a generic type can be checked with the `instanceof` type comparison operator as it is deemed safe at runtime.

Click here to view code image

```
IStack<Integer> iStack;
```

```
...
boolean isIntegerStack = iStack instanceof MyStack<Integer>;   // (2) OK.
```

The post-erasure code for (2) is equivalent to the following statement, where `My-Stack` is a subtype of `IStack` (i.e., the type of `iStack`), making the cast from `IStack` to `MyStack` safe.

Click here to view code image

```
boolean isIntegerStack = iStack instanceof MyStack;    // (2')
```

Given that `T` is a formal type parameter, the following code will not compile, as the arguments of the `instanceof` type comparison operator are non-reifiable types.

Click here to view code image

```
boolean isT      = obj instanceof T;                   // (3) Compile-time erro
boolean isTStack = obj instanceof MyStack<T>;          // (4) Compile-time erro
```

The post-erasure code for (2) and (3) is equivalent to the following statements. Again, casting the type `Object` to the types `Object` and `MyStack`, respectively, is not deemed safe at runtime.

Click here to view code image

```
boolean isT      = obj instanceof Object;              // (3')
boolean isTStack = obj instanceof MyStack;             // (4')
```

If we just wanted to determine that an instance was some stack, the instance test can be performed against the raw type or the unbounded wildcard parameterized type, as these types are reifiable:

Click here to view code image

```
boolean isRawStack = obj instanceof MyStack;
boolean isAnyStack = obj instanceof MyStack<?>;        // Preferable.
```

Implications for Casting

A non-reifiable type can lose important type information during erasure and the cast may not have the desired effect at runtime. A cast to a non-reifiable type is *generally* flagged as an *unchecked cast warning*, and the cast is *replaced by a cast to its erasure*. Again, the compiler permits casts to allow interoperability between legacy code and generic code—usually with a warning.

The following code shows why a warning is necessary. The reference value of a `Number` node, declared at (1), is assigned to a reference of type `Node<?>` at (2). This reference is cast to a `Node<String>` and its reference value is assigned to a reference of type `Node<String>` at (3). A `String` is set as data in the node at (4). The data is retrieved from the node via the `numNode` reference and assigned to a `Number` reference at (5).

Click here to view code image

```
Node<Number> numNode = new Node<>(20, null);         // (1)
Node<?> anyNode = numNode;                           // (2)
Node<String> strNode = (Node<String>) anyNode;       // (3) Unchecked cast warni
strNode.setData("Peekaboo");                         // (4)
Number num = numNode.getData();                      // (5) ClassCastException
```

The erasure of the assignment at (3) is equivalent to the following assignment, with the cast succeeding at runtime:

Click here to view code image

```
Node strNode = (Node) anyNode;                       // (3')
```

However, a `ClassCastException` occurs at (5) because a `String` cannot be assigned to a `Number`. The compiler warns of potential problems by issuing an unchecked cast warning at (3).

The types `Node<String>` and `Node<Number>` are unrelated. That is the reason why the `Number` node in the above example was compromised by going through a node of type `Node<?>`. As we would expect, a cast between unrelated types results in a compile-time error:

Click here to view code image

```
strNode = (Node<String>) numNode;                    // Compile-time error
```

If we are casting a generic supertype to a generic subtype, where the parameterization is identical, the cast is safe and no warning is issued:

Click here to view code image

```
// BiNode<E> is a subtype of MonoNode<E>.
MonoNode<String> monoStrNode = new BiNode<>("Hi", null, null);
BiNode<String> biStrNode = (BiNode<String>) monoStrNode; // Ok. No warning.
```

The method `castaway()` below shows examples of casting an `Object` reference that refers to a node of type `String`, declared at (2).

Click here to view code image

```
  //@SuppressWarnings("unchecked")         // (1) Suppress warnings at (4),(6),(
  public static void castaway() {
    Object obj = new Node<>("one", null);                 // (2)
    Node<String> node1 = obj;                             // (3) Compile-time error
    Node<String> node2 = (Node<String>) obj;              // (4) Unchecked cast
    Node<String> node3 = (Node<?>) obj;                   // (5) Compile-time error
    Node<String> node4 = (Node<String>)(Node<?>) obj;     // (6) Unchecked cast
    Node<String> node5 = (Node) obj;                      // (7) Unchecked conversi
    Node<?> node6 = (Node) obj;                           // (8) OK.
    Node<?> node7 = (Node<?>)obj;                         // (9) OK.
  }
```

It is instructive to see what warnings and errors are issued by the compiler. The compile-time error at (3) is due to incompatible types: An `Object` cannot be assigned to a `Node<String>` reference. The compiler issues an *unchecked cast warning* at (4) because of the cast from an `Object` to the concrete parameterized type `Node<String>`. The compile-time error at (5) is due to incompatible types: A `Node<?>` cannot be assigned to a `Node<String>` reference. There are two casts at (6): An `Object` is cast to `Node<?>`, which in turn is cast to `Node<String>`. The cast to `Node<?>` is permitted, but the second cast results in an unchecked cast warning. The compiler issues an *unchecked conversion warning* at (7), since a raw type (`Node`) is being assigned to a parameterized type (`Node<String>`). (8) and (9) show that casting to the raw type or to the unbounded wildcard is always permitted, since both types are reifiable.

If the annotation `@SuppressWarnings("unchecked")` at (1) is uncommented, the unchecked warnings at (4), (6), and (7) in the method `castaway()` will be suppressed (§25.5, p. 1582). Use of this annotation is recommended when we *know* that unchecked cast warnings are inevitable in a language construct (a type declaration, a field, a method, a parameter, a constructor, a local variable). Any unchecked warnings reported by the compiler are those that were *not* documented using this annotation. The use of an unbounded wildcard is recommended in casts, rather than using raw types, as it provides for stricter type checking.

Implications for Arrays

Array store checks are based on the element type being a reifiable type, in order to ensure that subtype covariance between array types is not violated at runtime. In the code below, the element type of the array is `String` and the array store check at (1) disallows the assignment, resulting in an `ArrayStoreException` because the reference value of a `Double` cannot be stored in a `String` reference.

Click here to view code image

```
String[] strArray = new String[] {"Hi", "Hello", "Howdy"};
Object[] objArray = strArray; // String[] is a subtype of Object[]
objArray[0] = 2020.5;         // (1) ArrayStoreException
```

We cannot instantiate a formal type parameter, nor can we create an array of such a type:

Click here to view code image

```
// T is a formal type parameter.
T t = new T();                // Compile-time error!
T[] anArray = new T[10];      // Compile-time error!
```

It is also not possible to create an array whose element type is a concrete or a bounded wildcard parameterized type:

Click here to view code image

```
// An array of Lists of String
List<String>[] list1 = {                           // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<String>[] list2 = new List<String>[] {        // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

// An array of Lists of any subtype of Number
List<? extends Number>[] list3
               = new List<? extends Number>[] {    // Compile-time error

  Arrays.asList(20.20, 60.60), Arrays.asList(1948, 1949)
};
```

Unbounded wildcard parameterized types are allowed as element types because these types are essentially equivalent to the raw types (p. 584):

Click here to view code image

```
List<?>[] list4 = {
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<?>[] list5 = new List<?>[] {
  Arrays.asList(20.20, 60.60), Arrays.asList(1978, 1981)
};
List[] list6 = list5;
```

Note that we can always declare a *reference* of a non-reifiable type. It is creating arrays of these types that is not permitted.

Click here to view code image

```
class MyIntList extends ArrayList<Integer> {}     // A reifiable subclass.

// Client code
```

```
List<Integer>[] arrayOfLists = new MyIntList[5];  // Array of Lists of Integer
List<Integer[]> listOfArrays = new ArrayList<>(); // List of Arrays of Integer
```

The class `MyStack<E>` in Example 11.10, p. 598, implements a method to convert a stack to an array:

Click here to view code image

```
// Copy to array as many elements as possible.
public E[] toArray(E[] toArray) {                              // (11)
  Node<E> thisNode = tos;
  for (int i = 0; thisNode != null && i < toArray.length; i++) {
    toArray[i] = thisNode.getData();
    thisNode = thisNode.getNext();
  }
  return toArray;
}
```

Note that the array is passed as a parameter because we cannot create an array of the parameter type, as the following version of the method shows:

Click here to view code image

```
public E[] toArray2() {
  E[] toArray = new E[numOfElements];             // Compile-time error
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The third version below uses an array of `Object`. The cast is necessary in order to be compatible with the return type. However, the cast is to a non-reifiable type, resulting in an unchecked cast warning:

Click here to view code image

```
public E[] toArray3() {
  E[] toArray = (E[])new Object[numOfElements];  // (1) Unchecked cast warning
  int i = 0;

  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The method implementation above has a serious problem, even though the code compiles. We get a `ClassCastException` at (2) below because we cannot assign the reference value of an `Object[]` to an `Integer[]` reference:

Click here to view code image

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
```

```
Integer[] intArray = intStack.toArray3();         // (2) ClassCastException
```

The final and correct version of this method uses *reflection* to create an array of the right element type:

Click here to view code image

```
@SuppressWarnings("unchecked")
public E[] toArray4(E[] toArray) {
  if (toArray.length != numOfElements) {
    toArray = (E[])java.lang.reflect.Array.newInstance(              // (3)
                          toArray.getClass().getComponentType(),
                          numOfElements);          // Suppressed unchecked warn
  }
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The method is passed an array whose element type is determined through reflection, and an array of this element type (and right size) is created at (3). The method `newInstance()` of the `Array` class creates an array of specified element type and size. The element type is looked up through the class literal of the array supplied as an argument. The unchecked cast warning is suppressed because we know it is unavoidable. We will not go into the nitty-gritty details of using reflection here.

The client code now works as expected. We pass an array of zero length, and let the method create the array.

Click here to view code image

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
Integer[] intArray = intStack.toArray4(new Integer[0]);            // OK.
```

The next example demonstrates the danger of casting an array of a reifiable type to an array of a non-reifiable type. An array of the raw type `List` (reifiable type) is created at (1), and cast to an array of `List<Double>` (non-reifiable type). The cast results in an unchecked cast warning. The first element of the array of `List<Double>` is initialized with a list of `Double` at (2). The reference value of this array is assigned to a reference of type `List<? extends Number>` at (3). Using this reference, the list of `Double` in the first element of the array is replaced with a list of `Integer` at (4). Using the alias `arrayOfListsOfDouble` of type `List<Double>[]`, the first element in the first list of the array (an `Integer`) is assigned to a `Double` reference. Since the types are incompatible, a `ClassCastException` is thrown at (5). Note that the array store check at (4) succeeds because the check is against the reified element type of the array, `List`, and not `List<Double>`.

Click here to view code image

```
List<Double>[] arrayOfListsOfDouble
               = (List<Double>[]) new List[1];   // (1) Unchecked cast warnin
arrayOfListsOfDouble[0] = Arrays.asList(10.10);   // (2) Initialize
List<? extends Number>[] arrayOfListsOfExtNums = arrayOfListsOfDouble; // (3)
arrayOfListsOfExtNums[0] = Arrays.asList(10);     // (4) Array storage check o
Double firstOne = arrayOfListsOfDouble[0].get(0); // (5) ClassCastException!
```

Implications for Non-Reifiable Variable Arity Parameter

Because variable arity parameters are treated as arrays, generics have implications for non-reifiable variable arity parameters (`T ... varargs`). Most of the workarounds for arrays are not applicable, as array creation is implicit for variable arity parameters. In a method declaration with a non-reifiable variable arity parameter, the compiler flags a warning about possible *heap pollution* from using a non-reifiable variable arity type. Heap pollution occurs when a reference of a parameterized type refers to an object that is not of the parameterized type. This can only occur when both the compiler and the JVM cannot guarantee the validity of an operation involving parameterized types. The compiler issues a warning about potential heap pollution, and the JVM normally throws an appropriate exception.

The method `asStack()` below has a variable arity parameter at (1) whose type is a non-reifiable type `T`. The method pushes the specified `elements` on to the specified `stack`. The compiler issues a possible heap pollution warning at (1).

Click here to view code image

```
public static <T> void
  asStack(MyStack<T> stack, T...elements) { // (1) Possible heap pollution war
  for (T element : elements) { stack.push(element); }
}
```

In a method call, implicit creation of a *generic array with the variable arity parameter* results in an *unchecked generic array creation warning* —and type-safety is no longer guaranteed. The method above is called by the client code below at (4). The idea is to initialize a stack of stacks of `Integer` with a stack of `Integer`. An implicit generic array (`new MyStack[] { intStack }`) is created by the compiler, which is passed in the method call at (4). The compiler also issues an *unchecked array creation warning*, but the code compiles and runs without any problems.

Click here to view code image

```
/* Client code */
// (2) Create a stack of stacks of Integer:
MyStack<MyStack<Integer>> stackOfIntStacks = new MyStack<>();

// (3) Create a stack of Integer:
MyStack<Integer> intStack = new MyStack<>();
intStack.push(2019); intStack.push(2020);

// Initializes the stack of stacks with the stack of Integer.
MyStack.asStack(stackOfIntStacks, intStack); // (4) Unchecked array creation!
intStack = stackOfIntStacks.pop();      // (5) Pop the stack of stacks of Inte
int tos = intStack.pop();               // (6) Pop the stack of Integer.
assert tos == 2020;
```

The implicit array passed as an argument is available as an array of a *non-reifiable type* in the body of the method `asStack()`. The integrity of this array can be compromised by making the array store check report a false positive at runtime—that is, succeed when the store operation should normally fail, thereby resulting in heap pollution. This is demonstrated by the method declaration below, in the assignment statement at (1a), where the contents of the `elements` array are changed before they are copied to the specified `stack`. The compiler issues a possible heap pollution warning at (1) and an unchecked cast warning at (1a).

Click here to view code image

```
public static <T> void asStackMalicious(
         MyStack<T> stack, T... elements) { // (1) Possible heap pollution war
  // Compromise the elements array:
  MyStack<Double> doubleStack = new MyStack<>();
  doubleStack.push(20.20);
  elements[0] = (T) doubleStack;            // (1a) Unchecked cast warning!

  // Copy from elements array:
  for (T element : elements) { stack.push(element); }
}
```

A partial erasure for the method `asStackMalicious()` is shown below.

Click here to view code image

```
public static void asStackMalicious(MyStack stack, Object...elements) {
  // Compromise the elements array:
  MyStack doubleStack = new MyStack();
  doubleStack.push(Double.valueOf(20.20));
  elements[0] = (Object) doubleStack;                      // (1b)
  for (Object element : elements) { stack.push(element); } // (1c)
...
}
```

Note that the *cast* at (1b) succeeds for *any* object at runtime, as any object can be cast to `Object`. The assignment at (1b) succeeds if the array store check succeeds.

If we now call the method `asStackMalicious()`, instead of the method `asStack()` at (4) in the client code above, the code compiles with a generic array creation warning as before.

Click here to view code image

```
MyStack.asStackMalicious(stackOfIntStacks, intStack); // (4') Unchecked warnin
```

At runtime, the reference `elements` in the method `asStackMalicious()` refers to the implicit array created in the call—that is, `new MyStack[] { intStack }`. The signature of the method call at runtime is equivalent to:

Click here to view code image

```
asStackMalicious(MyStack, MyStack[])
```

At runtime, the actual type of the `stack` parameter is `MyStack` and the actual type of the `elements` array parameter is `MyStack[]`. The element type of a stack has been erased. The references `doubleStack` and `elements[0]` both have the runtime type `MyStack`. When the code is run, the array store check succeeds at (1b), and `element[0]` now refers to a stack of `Double`. In the loop at (1c), this stack of `Double` is pushed on the stack of stacks of `Integer` referred to by the formal parameter `stack`. Heap pollution is now a fact.

After return from the call at (4) in the client code, the assignment at (5) also succeeds, as this is an assignment of a `MyStack` to a reference of the same type after erasure. The reference `intStack` now refers to a stack of `Double`. The error is only discovered after a `ClassCastException` is thrown at (6) because the `Double` that is popped from the stack of `Integer` cannot be converted and assigned to an `Integer`.

The general rule is to avoid the variable arity parameter in methods where the parameter is of a non-reifiable type. No matter what, it is the responsibility of the method declaration to ensure that the non-reifiable variable arity parameter is handled in a type-safe manner, and then to use one of the two annotations below to suppress unchecked warnings:

Click here to view code image

```
@SuppressWarnings("unchecked")                                // (1)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }

@SafeVarargs                                                  // (2)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }
```

The method `asStack()` handles the non-reifiable variable arity parameter `elements` in a type-safe way, so suppressing the unchecked warning is justified. The annotation at (1) suppresses *all* unchecked warnings in the method declaration, but it does *not* suppress the unchecked generic array creation warning at the call sites. The annotation at (2) suppresses all unchecked warnings in the method declaration, as well as the unchecked generic array creation warning at the call sites. However, the `@SafeVarargs` annotation (§25.5, p. 1582) is only applicable to a variable arity method or constructor, and the method cannot be overridden—that is, the method must be either `static`, `private`, or `final`.

Implications for Exception Handling

When an exception is thrown in a `try` block, it is matched against the parameter of each `catch` block that is associated with the `try` block. This test is similar to the instance test, requiring reifiable types. The following restrictions apply, and are illustrated in Example 11.17:

- A generic type cannot extend the `Throwable` class.
- A parameterized type cannot be specified in a `throws` clause.
- The type of the parameter of a `catch` block must be a reifiable type, and it must also be a subtype of `Throwable`.

Example 11.17 *Restrictions on Exception Handling*

Click here to view code image

```
// File: ExceptionErrors.java

// (1) A generic class cannot extend Exception:
class MyGenericException<T> extends Exception { }        // Compile-time error

public class ExceptionErrors {

  // (2) Cannot specify parameterized types in throws clause:
  public static void main(String[] args)
                        throws MyGenericException<String> { // Compile-time error
    try {
      throw new MyGenericException<String>();
    } // (3) Cannot use parameterized type in catch block:
      catch (MyGenericException<String> e) {              // Compile-time error
      e.printStackTrace();
```

```
        }
      }
    }
```

However, *type parameters* are allowed in the `throws` clause, as shown in Example 11.18. In the declaration of the `MyActionListener` interface, the method `doAction()` `can` throw an exception of type `E`. The interface is implemented by the class `FileAction`, that provides the actual type parameter (`FileNotFoundException`) and implements the `doAction()` method with this actual type parameter. All is aboveboard, as only reifiable types are used for exception handling in the class `FileAction`.

Example 11.18 *Type Parameter in* `throws` *Clause*

Click here to view code image

```
public interface MyActionListener<E extends Exception> {
  public void doAction() throws E;       // Type parameter in throws clause
}
______________________________________________________________________________
import java.io.FileNotFoundException;

public class FileAction implements MyActionListener<FileNotFoundException> {

  @Override public void doAction() throws FileNotFoundException {
     throw new FileNotFoundException("Does not exist");
  }

  public static void main(String[] args) {
    FileAction fileAction = new FileAction();
    try {
      fileAction.doAction();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Implications for Nested Classes

Nested classes and interfaces can be declared as generic types, as shown in Example 11.19. All nested generic classes, except anonymous classes, can specify formal type parameters in their declaration, as at (2) through (6). Anonymous classes do not have a name, and a class name is required to declare a generic class and specifyits formal type parameters. Thus an anonymous class can only be declared as a parameterized type, where the actual type parameters are either specified in the anonymous class expression, as at (7) through (9), or can be inferred from the context when the diamond operator is used, as at (10).

Example 11.19 *Generic Nested Types*

Click here to view code image

```
class GenericTLC<A> {             // (1) Top-level class

  static class SMC<B> {/*...*/}   // (2) Static member class

  interface SMI<C> {/*...*/}      // (3) Static member interface

  class NSMC<D> {/*...*/}         // (4) Non-static member (inner) class

  void nsm() {
    class NSLC<E> {/*...*/}       // (5) Local (inner) class in non-static con
  }

  static void sm() {
    class SLC<F> {/*...*/}        // (6) Local (inner) class in static context
  }

  // Anonymous classes as parameterized types:
  SMC<A> xsf = new SMC<A>() {     // (7) In non-static context.
    /*...*/                       //     A is type parameter in top-level clas
  };
  SMC<Integer> nsf = new SMC<Integer>() {           // (8) In non-static conte
    /*...*/
  };
  static SMI<String> sf = new SMI<String>() {       // (9) In static context
    /*...*/
  };
  SMC<String> nsf1 = new SMC<>() {                  // (10) Using diamond oper
    /*...*/
  };
}
```

The type parameter names of a generic nested class can hide type parameter names in the enclosing context (see (2) in Example 11.20). Only a non-static nested class can use the type parameters in its enclosing context, as type parameters cannot be referenced in a static context.

Example 11.20 also illustrates instantiating generic nested classes. As a static member class does not have an outer instance, only its simple name is parameterized, and not the enclosing types, as shown by the code at (6). As a non-static member class requires an outer instance, any generic enclosing types must also be parameterized and instantiated, as shown by the code at (7). See §9.3, p. 502, for the syntax used in instantiating nested classes.

Example 11.20 *Instantiating Generic Nested Classes*

Click here to view code image

```
public class ListPool<T> {       // (1) Top-level class

  static class MyLinkedList<T> { // (2) Hiding type parameter in enclosing con
    T t;                         // T refers to (2)
  }

  class Node<E> {                // (4) Non-static member (inner) class
    T t;                         // T refers to (1)
```

```
      E e;
    }

    public static void main(String[] args) {
      // (5) Instantiating a generic top-level class:
      ListPool<String> lp = new ListPool<>();

      // (6) Instantiating a generic static member class:
      ListPool.MyLinkedList<String> list = new ListPool.MyLinkedList<>();

      // (7) Instantiating a generic non-static member class:
      ListPool<String>.Node<Integer> node1 = lp.new Node<>();
      ListPool<String>.Node<Double> node2 = lp.new Node<>();
      ListPool<Integer>.Node<String> node3
                            = new ListPool<Integer>().new Node<>();
    }
  }
```

Other Implications

Enum Types

Because of the way enum types are implemented using the `java.lang.Enum` class, we cannot declare a generic enum type:

Click here to view code image

```
enum CoinToss<C> { HEAD, TAIL; }                  // Compile-time error!
```

An enum type can implement a parameterized interface, just like a non-generic class can. Example 11.21 shows the enum type `TripleJump` that implements the `Comparator<TripleJump>` interface.

Example 11.21 *Enum Implements Parameterized Generic Interface*

Click here to view code image

```
enum TripleJump implements java.util.Comparator<TripleJump> {
  HOP, STEP, JUMP;

  @Override public int compare(TripleJump a1, TripleJump a2) {
    return a1.compareTo(a2);
  }
}
```

Class Literals

Objects of the class `Class<T>` represent classes and interfaces at runtime. For example, an instance of the `Class<String>` represents the type `String` at runtime.

A class literal expression (using `.class` notation) can only use reifiable types as type parameters, as there is only one class object created for each reifiable type.

Click here to view code image

```
Class<Node> class0 = Node<Integer>.class;// Non-reifiable type. Compile-time e
Class<Node> class1 = Node.class;        // Reifiable type. Ok.
```

The `getClass()` method of the `Object` class also returns a `Class` object. The actual result type of this object is `Class<? extends |T|>` where `|T|` is the erasure of the static type of the expression on which the `getClass()` method is called. The following code shows that all invocations of the generic type `Node<T>` are represented by a single class literal:

Click here to view code image

```
Node<Integer> intNode = new Node<>(2019, null);
Node<String> strNode = new Node<>("Hi", null);
Class<?> class2 = strNode.getClass();
Class<?> class3 = intNode.getClass();
assert class1 == class2;
assert class2 == class3;
```

Review Questions

11.11 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.*;

public class RQ100_00 {
  public static void main(String[] args) {
    List<String> lst1 = new ArrayList<>();
    List<Integer> lst2 = new ArrayList<>();
    List<List<Integer>> lst3 = new ArrayList<>();
    System.out.print(lst1.getClass() + ", ");
    System.out.print(lst2.getClass() + ", ");
    System.out.println(lst3.getClass());
  }
}
```

Select the one correct answer.

a. Click here to view code image

```
class java.util.ArrayList<String>, class java.util.ArrayList<Integer>,
java.util.ArrayList<List<Integer>>
```

b. Click here to view code image

```
class java.util.ArrayList, class java.util.ArrayList, class
java.util.ArrayList
```

c. Click here to view code image

```
class java.util.List, class java.util.List, class java.util.List
```

d. Click here to view code image

```
class java.util.List<String>, class java.util.List<Integer>, class
java.util.List<List<Integer>>
```

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

11.12 Which method header can be inserted at (1) so that the program will compile and runs without errors?

Click here to view code image

```
import java.util.*;
public class RQ100_02 {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList("Java", "only", "promotes", "fun");
    Collection<String> resultList = delete4LetterWords(lst);
  }

  // (1) INSERT METHOD HEADER HERE
  {
    Collection<E> permittedWords = new ArrayList<>();
    for (E word : words) {
      if (word.length() != 4) permittedWords.add(word);
    }
    return permittedWords;
  }
}
```

Select the one correct answer.

a. Click here to view code image

```
static <E extends CharSequence>
Collection<? extends CharSequence> delete4LetterWords(Collection<E> wor
```

b. Click here to view code image

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<E> words)
```

c. Click here to view code image

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<? extends CharSequence> wor
```

d. Click here to view code image

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<? extends CharSequence> words)
```

e. Click here to view code image

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

f. Click here to view code image

```
static <E super CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

11.13 Which method declarations cannot be inserted independently at (2) to overload the method at (1)?

Click here to view code image

```
import java.util.*;

public class RQ_Overloading {

  static <T> void overloadMe(List<T> s1, List<T> s2) {} // (1)
  // (2) INSERT METHOD DECLARATIONS HERE
}
```

Select the two correct answers.

a. Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, List<T> s2) {}
```

b. Click here to view code image

```
static <T> void overloadMe(List<T> s1, List<? extends T> s2) {}
```

c. Click here to view code image

```
static <T> void overloadMe(List<T> s1, Collection<? super T> s2) {}
```

d. Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, Collection<? super T> s2)
```

e. Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, List<? super T> s2) {}
```

f. Click here to view code image

```
static <T> void overloadMe(List<? extends T> s1, List<? super T> s2) {}
```

11.14 Which method declaration can be inserted at (1) so that the program will compile without warnings?

Click here to view code image

```
import java.util.*;
public class RQ100_87 {
  public static void main(String[] args) {
    List raw = new ArrayList();
    raw.add("2020");
    raw.add(2021);
    raw.add("2022");
    justDoIt(raw);
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

Select the one correct answer.

a. `static void justDoIt(List<Integer> lst) {}`

b. `static void justDoIt(List<?> lst) {}`

c. `static <T> void justDoIt(List<T> lst) {}`

d. None of the above

11.15 Which of the following statements are true about the following code?

Click here to view code image

```
class SupC<T> {
  public void set(T t) {/*...*/}        // (1)
  public T get() {return null;}         // (2)
}

class SubC1<M,N> extends SupC<M> {
  public void set(N n) {/*...*/}        // (3)
  public N get() {return null;}         // (4)
}

class SubC2<M,N extends M> extends SupC<M> {
  public void set(N n) {/*...*/}        // (5)
  public N get() {return null;}         // (6)
}
```

Select the four correct answers.

a. The method at (3) overloads the method at (1).

b. The method at (3) overrides the method at (1).

c. The method at (3) will fail to compile.

d. The method at (4) overloads the method at (2).

e. The method at (4) overrides the method at (2).

f. The method at (4) will fail to compile.

g. The method at (5) overloads the method at (1).

h. The method at (5) overrides the method at (1).

i. The method at (5) will fail to compile.

j. The method at (6) overloads the method at (2).

k. The method at (6) overrides the method at (2).

l. The method at (6) will fail to compile.

11.16 Which types cannot be declared as generic types? Select the three correct answers.

a. Enum types

b. Static member classes

c. Any subclass of `Throwable`

d. Nested interfaces

e. Anonymous classes

f. Non-static member classes

g. Local classes

11.17 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;

public class CastAway {
  public static void main(String[] args) {
    Object obj = new ArrayList<Integer>();         // (1)
    List<?>       list1 = (List<?>) obj;           // (2)
    List<?>       list2 = (List) obj;              // (3)
    List          list3 = (List<?>) obj;           // (4)
    List<Integer> list4 = (List) obj;              // (5)
    List<Integer> list5 = (List<Integer>) obj;     // (6)
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without any unchecked warnings. It will run with no output and terminate normally.

c. The program will compile without any unchecked warnings. When run, it will throw an exception.

d. The program will compile, but with unchecked warnings. It will run with no output and terminate normally.

e. The program will compile, but with unchecked warnings. When run, it will throw an exception.

11.18 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;
public class InstanceTest2 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    Set<Double> doubleSet = new HashSet<>();
    List<?>          list = intList;
    Set<?>            set = doubleSet;

    scuddle(intList);
    scuddle(doubleSet);
    scuddle(list);
    scuddle(set);
  }
  private static void scuddle(Collection<?> col) {
    if (col instanceof List<?>) {
      System.out.println("I am a list.");
    } else if (col instanceof Set<?>) {
      System.out.println("I am a set.");
    }
  }
}
```

Select the one correct answer.

a. The method `scuddle()` will fail to compile.

b. The method `main()` will fail to compile.

c. The program will compile, but with an unchecked warning in method `scuddle()`. It will run and terminate normally with the following output:

```
I am a list.
I am a set.
I am a list.
I am a set.
```

d. The program will compile, but with an unchecked warning in the method `main()`. When run, it will throw an exception.

e. The program will compile without any unchecked warnings. It will run and terminate normally, with the following output:

```
I am a list.
I am a set.
I am a list.
I am a set.
```

f. The program will compile without any unchecked warnings. It will run and terminate normally, with the following output:

```
I am a list.
I am a set.
```

g. None of the above

11.19 Which statement is true about the following code?

Click here to view code image

```
public class GenArrays {
  public static <E> E[] copy(E[] srcArray) {
    E[] destArray = (E[]) new Object[srcArray.length];
    int i = 0;
    for (E element : srcArray) {
      destArray[i++] = element;
    }
    return destArray;
  }

  public static void main(String[] args) {
    String[] sa = {"9", "1", "1" };
    String[] da = GenArrays.copy(sa);
    System.out.println(da[0]);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but with an unchecked warning. When run, it will print `9`.

c. The program will compile, but with an unchecked warning.

When run, it will throw an exception.

d. The program will compile without any unchecked warnings. When run, it will print `9`.

e. The program will compile without any unchecked warnings. When run, it will throw an exception.

11.20 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;
public class GenVarArgs {
  public static <T> void doIt(List<T>... aols) {           // (1)
    for (int i = 0; i < aols.length; i++) {
      System.out.print(aols[i] + " ");
    }
  }

  public static void main(String... args) {                 // (2)
    List<String> ls1 = Arrays.asList("one", "two");
    List<String> ls2 = Arrays.asList("three", "four");
    List<String>[] aols = new List[] {ls1, ls2};            // (3)
    doIt(aols);                                             // (4)
  }
}
```

Select the one correct answer.

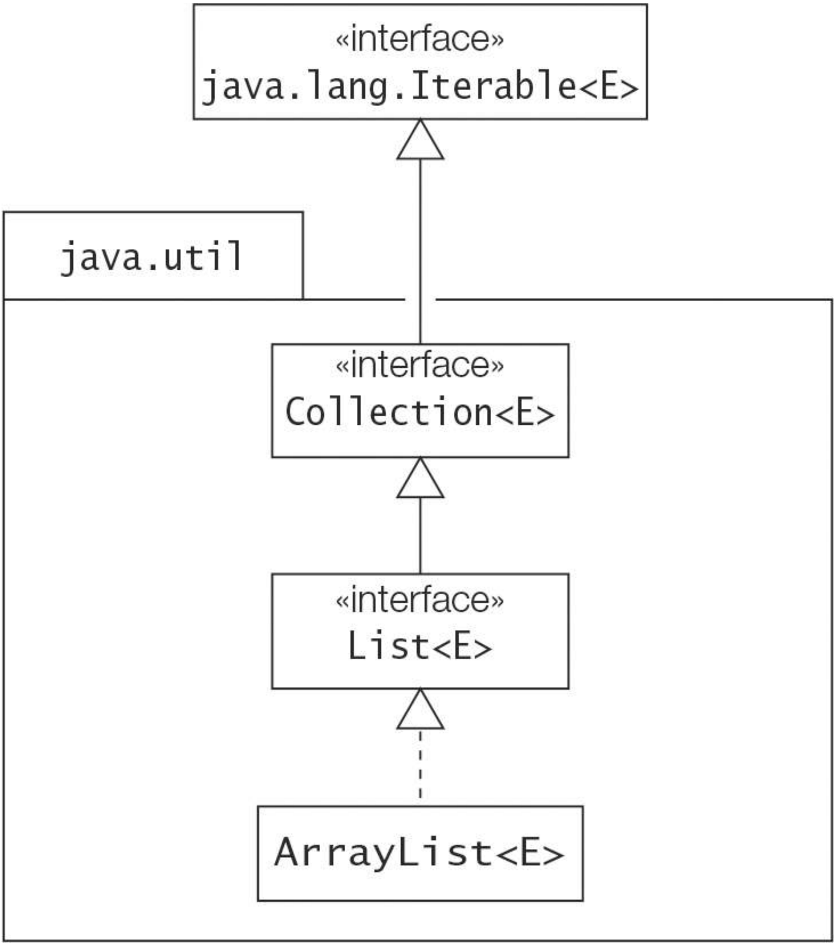
a. The program will fail to compile because of errors at (1).

b. The program will fail to compile because of errors at (2).

c. The program will fail to compile because of errors at (3).

d. The program will fail to compile because of errors at (4).

e. The program will compile with unchecked warnings. When run, it will print `[one, two] [three, four]`.


«interface»
java.lang.Iterable<E>
java.util
«interface»
Collection<E>
«interface»
List<E>
ArrayList<E>

Collections, Part I: `ArrayList<E>`

12

Chapter Topics

- Understanding the concept of a list as a collection
- The inheritance relationship between the `ArrayList<E>`

class, the `List<E>` interface, and the `Collection<E>` interface in the Java Collections Framework

- Declaring and using references of the `ArrayList<E>` type
- Creating unmodifiable lists
- Creating, modifying, querying, and traversing `ArrayList`s
- Interoperability between arrays and `ArrayList`s
- Comparison of arrays and `ArrayList`s

Java SE 17 Developer Exam Ob

[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements

§12.1, p. 644, to §12.8, p. 662.

- *Only* `ArrayList` *is covered in this chapter.*
- *For arrays, see §3.9, p. 117.*
- *For comparing elements, see §14.4, p. 761, and §14.5, p. 769.*
- *For list, set, map, and deque collections, see Chapter 15, p. 781.*

Java SE 11 Developer Exam O

[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods

§12.1, p. 644, to §12.8, p. 662.

- *Only* `ArrayList` *is covered in this chapter.*
- *For arrays, see §3.9, p. 117.*
- *For list, set, map, and deque collections, see Chapter 15, p. 781.*

A program manipulates data, so naturally, organizing and using data efficiently is important in a program. *Data structures* allow data to be organized in an efficient way. Java uses the term *collection* to mean a data structure that can maintain a group of objects so that the objects can be manipulated as a *single entity* or *unit*. Objects can be stored, retrieved, and manipulated as *elements* of a collection. The term *container* is also used in the literature for such data structures. Arrays are one example of such collections. Other examples include lists, sets, queues, and stacks, among many others.

The Java Collections Framework provides the support for collections in Java. This chapter only covers the core API of the `ArrayList<E>` class that implements dynamic lists. We will have more to say about the `ArrayList<E>` class when we discuss the other collections in the Java Collections Framework in Chapter 15, p. 781. Diving deep into the Java Collections Framework is a beneficial exercise that is highly recommended for all Java programmers.

As the collections in the Java Collections Framework are implemented as generic types, knowledge of at least the basics of generics in Java is essential to utilize these collections effectively (Chapter 11, p. 563).

12.1 Lists

Once an array is created, its length cannot be changed. This inflexibility can be a significant drawback when the amount of data to be stored in an array is not known a priori. In Java, the structures known as lists alleviate this shortcoming. Lists are collections that maintain their elements *in order* and can contain duplicates. The order of elements in a list is *positional order*, and individual elements can be accessed according to their position in the list. Each element, therefore, has a position in the list. A zero-based index can be used to access the element at the position designated by the index value, analogous to accessing elements in an array. However, unlike in an array, the position of an element in a list can change as elements are inserted or deleted from the list—that is, as the list changes dynamically.

Sorting implies ordering the elements in a collection according to some *ranking criteria*, usually based on the *values* of the elements. However, elements is an `ArrayList` are maintained in the order they are inserted in the list, known as the *insertion order*. The elements in such a list are therefore *ordered*, but they are *not* sorted, as it is not the values of the elements that determine their ranking in the list. Thus ordering does *not* necessarily imply sorting.

Overview of the Java Collections Framework

The `Collection<E>` interface in the `java.util` package (also known as the Java Collections Framework) defines the general operations that a collection should provide (see Figure 12.1). Note that the `Collection<E>` interface extends the `Iterable<E>` interface, so all collections in this framework can be traversed using the `for(:)` loop. Other subinterfaces in the Java Collections Framework augment this interface to provide specific operations for particular kinds of collections. The `java.util.List<E>` interface extends the `java.util.Collection<E>` interface with the operations necessary to maintain the collection as a list. In addition to the operations inherited from the `java.util.Collection<E>` interface, the `java.util.List<E>` interface defines operations that work specifically on lists: position-based access of the list elements, searching in a list, operations on parts of a list (called *open range-view* operations), and creation of customized iterators to iterate over a list. For methods used in this chapter, we will indicate which interface they are defined in. The impatient reader can refer to Chapter 15, p. 781, at any time for more details on these interfaces.

Figure 12.1 *Partial* `ArrayList` *Inheritance Hierarchy*

The *generic* class `java.util.ArrayList<E>` implements the `java.util.List<E>` interface. The type parameter `E` represents the type of the element in the list. Use of a generic type requires a *concrete* reference type to be substituted for the type parameter `E`. For example, the parameterized class `ArrayList<String>` is an `ArrayList` of `String`, where the type parameter `T` is substituted with the concrete class `String`.

The `ArrayList<E>` class is a dynamically resizable implementation of the `List<E>` interface using arrays (also known as *dynamic arrays*), providing fast random access (i.e., position-based access in constant time) and fast list traversal—very much like using an ordinary array. The `ArrayList<E>` class is not *thread-safe*; that is, its integrity can be jeopardized by concurrent access. The Java Collections Framework provides other implementations of the `List<E>` interface, but in most cases the `ArrayList<E>` implementation is the overall best choice for implementing lists.

12.2 Declaring References and Constructing `ArrayLists`

In the discussion that follows, we assume that any class or interface used from the `java.util` package has been imported with an appropriate `import` statement.

The code below illustrates how we can create an empty `ArrayList` of a specific element type, and assign its reference value to a reference:

Click here to view code image

```
ArrayList<String> palindromes = new ArrayList<String>(); // (1)
```

The element type is specified using angle brackets (<>). The reference `palindromes` can refer to any `ArrayList` whose element type is `String`. The type parameter `E` of the class `ArrayList` in Figure 12.1 is replaced by the concrete class `String`. The compiler ensures that the reference `palindromes` can only refer to an `ArrayList` whose elements are of type `String`, and any operations on this list via this reference are type-safe.

The simplest way to construct an `ArrayList` is to use the zero-argument constructor to create an empty `ArrayList`, as shown in the declaration above. The zero-argument constructor creates an empty list with the initial capacity of 10. The *capacity* of a list refers to how many elements it can contain at any given time, not how many elements are actually in the list (which is called the *size*). The capacity of a list and its size can change dynamically as the list is manipulated. The `Array-List<String>` created at (1) can only contain elements of type `String`.

The assignment in the declaration statement (1) is valid because the types on both sides are assignment compatible—an `ArrayList` of `String`. The reference `palindromes` can now be used to manipulate the `ArrayList<String>` that it denotes.

We can use the *diamond operator* (<>) in the `ArrayList` creation expression on the right-hand side of the declaration statement. In this particular context, the compiler can infer the element type of the `ArrayList` from the declaration of the reference type on the left-hand side.

Click here to view code image

```
ArrayList<String> palindromes = new ArrayList<>(); // Using the diamond operat
```

However, if the diamond operator is omitted, the compiler will issue an *unchecked conversion warning*, as shown at (2) in the next code snippet. A new `ArrayList` is created based on an `ArrayList` of `Integer` that is passed as an argument to the constructor. The `ArrayList` of `Integer` is created at (1). The reference `newList1` of type `ArrayList<String>` refers to an `ArrayList` whose element type is `Integer`, not `String`. The code at (2) compiles, but we get a `ClassCastException` at runtime at (3) when we retrieve an element from this list. The `get()` method call at (3) expects a `String` in the `ArrayList`, but gets an `Integer`. If the diamond operator is used, as shown at (4), the compiler reports a compile-time error, and the problem described at (3) cannot occur at runtime. By issuing an unchecked conversion warning at (2), the compiler alerts us to the fact that it cannot guarantee type-safety of the list created at (2).

Click here to view code image

```
ArrayList<Integer> intList = new ArrayList<>();        // (1) ArrayList of Int
intList.add(10); intList.add(100); intList.add(1000);
ArrayList<String> newList1 = new ArrayList(intList);   // (2) Unchecked conver
                                                       //     warning
System.out.println(newList1.get(0));                   // (3) ClassCastExcepti

ArrayList<String> newList2 = new ArrayList<>(intList); // (4) Compile-time err
```

Best practices advocate *programming to an interface*. In practical terms, this means using references of an interface type to manipulate objects of a concrete class that implement this interface. Since the class `java.util.ArrayList<E>` implements the `java.util.List<E>` interface, the declaration at (1) can be written as shown in the next code snippet. This declaration is valid, since the reference value of a subtype object (`ArrayList<String>`) can be assigned to a reference of its supertype (`List<String>`).

Click here to view code image

```
List<String> palindromes = new ArrayList<>();    // (2) List<String> reference
```

This best practice provides great flexibility in substituting other objects for a task when necessary. The current concrete class can easily be replaced by another concrete class that implements the same interface. Only code creating objects needs to be changed. As it happens, the Java Collections Framework provides another implementation of lists: the `java.util.LinkedList<E>` class, which also implements the `List<E>` interface. If this class is found to be more conducive for maintaining palindromes in a list, we need simply change the name of the class in declaration (2), and continue using the reference `palindromes` in the program:

Click here to view code image

```
List<String> palindromes = new LinkedList<>(); // Changing implementation.
```

The `ArrayList<E>` class also provides a constructor that allows an empty `ArrayList` to be created with a specific initial capacity.

Click here to view code image

```
List<String> palindromes = new ArrayList<>(20); // Initial capacity is 20.
```

The `ArrayList` class provides the `add(E)` method to append an element to the end of the list. The new element is added after the last element in the list, thereby increasing the list size by 1.

Click here to view code image

```
palindromes.add("level"); palindromes.add("Ada"); palindromes.add("kayak");
System.out.println(palindromes);
```

The print statement calls the `toString()` method in the `ArrayList<E>` class to print the elements in the list. This `toString()` method applies the `toString()` method of the individual elements to create a text representation in the following default format:

```
[level, Ada, kayak]
```

A third constructor allows an `ArrayList` to be constructed from another collection. The following code creates a list of words from a list of palindromes. The order of the elements in the new `ArrayList<String>` is the same as that in the `Array-List<String>` that was passed as an argument in the constructor.

Click here to view code image

```
List<String> wordList = new ArrayList<>(palindromes);
System.out.println(wordList); // [level, Ada, kayak]
wordList.add("Naan");
System.out.println(wordList); // [level, Ada, kayak, Naan]
```

The next examples illustrate the creation of empty lists of different types of elements. The compiler ensures that operations on the `ArrayList` are type-safe with respect to the element type. Declaration (3) shows how we can create nested list structures (i.e., a list of lists), analogous to an array of arrays. Note that the diamond operator is not nested at (3). Declaration (4) shows that the element type cannot be a primitive type; rather, it must be a reference type.

Click here to view code image

```
List<StringBuilder> synonyms    = new ArrayList<>(); // List of StringBuilder
List<Integer> attendance        = new ArrayList<>(); // List of Integer
List<List<String>> listOfLists = new ArrayList<>(); // (3) List of List of Str
List<int> frequencies           = new ArrayList<>(); // (4) Compile-time error!
```

When comparing arrays and `ArrayList`s, there is one other significant difference that concerns the subtype relationship.

Click here to view code image

```
Object[] objArray = new String[10];                    // (5) OK!
```

In declaration (5), since `String` is a subtype of `Object`, `String[]` is a subtype of `Object[]`. Thus we can manipulate the array of `String` using the `objArray` reference.

Click here to view code image

```
objArray[2] = "Green";                         // (6) OK!
objArray[1] = Integer.valueOf(2016);           // ArrayStoreException!
```

The preceding assignment requires a runtime check to guarantee that the assignment is type compatible. Otherwise, an `ArrayStoreException` is thrown at runtime.

For the `ArrayList<E>`, the following declarations will not compile:

Click here to view code image

```
ArrayList<Object> objList1 = new ArrayList<String>();// (7) Compile-time error
List<Object> objList2 = new ArrayList<String>();     // (8) Compile-time error
```

Although `String` is a subtype of `Object`, it is not the case that an `ArrayList<String>` is a subtype of `ArrayList<Object>`. If this were the case, we could use the `objList1` reference to add other types of objects to the `ArrayList` of `String`, thereby jeopardizing its type-safety. Since there is no information about the element type `E` available at runtime to carry out a type compatibility check, as in the case of arrays, the subtype relationship is not allowed at (7). For the same reason, (8) will also not compile: `ArrayList<String>` is not a subtype of `List<Object>`. In general, the *subtype covariant relationship* does not hold for generic types. The Java language provides *wildcards* to overcome this restriction (§11.4, p. 579).

The `ArrayList<E>` constructors are summarized here:

Click here to view code image

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

The zero-argument constructor creates a new, empty `ArrayList` with an initial capacity of 10.

The second constructor creates a new, empty `ArrayList` with the specified initial capacity.

The third constructor creates a new ArrayList containing the elements in the specified collection. The declaration of the parameter c essentially means that parameter c can refer to any collection whose element type is E or a subtype of E. The new ArrayList<E> will retain any duplicates. The ordering in the ArrayList<E> will be determined by the traversal order of the iterator for the collection passed as an argument.

In a constructor call, the element type of the list is specified

> enclosed in angle brackets or by the diamond operator after the class name if it is to be inferred by the compiler. A raw `ArrayList` is created if the angle brackets are omitted, and the compiler will issue an unchecked warning.

Creating Unmodifiable Lists

Unmodifiable collections are useful to prevent a collection from accidently being modified, as doing so might cause the program to behave incorrectly. Such collections are also stored efficiently, as no bookkeeping is required to support any further modifications and data in the collection can be packed more densely since it can never change.

Here we look at how to create unmodifiable lists. Later we will discuss unmodifiable sets (§15.4, p. 804) and unmodifiable maps (§15.8, p. 832), and we will also contrast *unmodifiable collections* with *unmodifiable views of collections* (§15.11, p. 856).

The `List<E>` interface provides generic static methods to create *unmodifiable* lists that have the following characteristics:

- An unmodifiable list cannot be modified *structurally*; for example, elements cannot be added, removed, replaced, or sorted in such a list. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the elements themselves are mutable, the elements may appear modified.
- Although duplicates are allowed, unmodifiable lists do not allow `null` elements, and will result in a `NullPointerException` if an attempt is made to create them with the `null` elements.
- The order of the elements in an unmodifiable list is the same as the order of the arguments or the order of the elements in the array of the variable arity argument of the static method.
- An unmodifiable list can be serialized if its elements are serializable (§20.5, p. 1261).

Click here to view code image

```
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5,
                      E e6, E e7, E e8, E e9, E e10)
```

> The `of()` method is overloaded. The 11 overloaded methods are fixed-argument methods for accepting 0 to 10 arguments. They return an unmodifiable list containing the number of elements specified. They throw a `NullPointerException`, if an element is `null`. These overloaded methods are convenient for creating short lists.
>
> Click here to view code image
>
> ```
> @SafeVarargs static <E> List<E> of(E... elements)
> ```
>
> This variable arity method returns an unmodifiable list containing an arbitrary number of elements specified by its variable arity argument. It throws a `Null-PointerException`, if an element is `null` or if the array of the variable arity parameter is `null`.
> The `@SafeVarargs` annotation suppresses the heap pollution warning in the method declaration and also the unchecked generic array creation warning at the call sites (§25.5, p. 1585).
>
> Click here to view code image
>
> ```
> static <E> List<E> copyOf(Collection<? extends E> collection)
> ```
>
> This generic method returns an unmodifiable list containing the elements of the specified collection, in its iteration order. The specified collection must not be `null`, and it must not contain any `null` elements—otherwise, a `NullPointerException` is thrown. If the specified collection is subsequently modified, the returned list will not reflect such modifications.

The code below shows that a list created by the `List.of()` method cannot be modified. The list returned is also not an instance of `ArrayList`.

Click here to view code image

```
List<String> list = List.of("Tom", "Dick", "Harriet");
//  list.add("Harry");                         // UnsupportedOperationExcepti
//  list.remove(2);                            // UnsupportedOperationExcepti
//  list.set(0, "Tommy");                      // UnsupportedOperationExcepti
System.out.println(list);                      // [Tom, Dick, Harriet]
System.out.println(list instanceof ArrayList); // false
```

The `List.of()` method does not allow `null` elements:

Click here to view code image

```
List<String> coinList = List.of("nickel", "dime", null); // NullPointerExcepti
```

For arguments up to 10, an appropriate fixed-arity `List.of()` method is called. For more than 10 arguments, the variable arity `List.of(E...)` method is called, passing an implicitly created array that contains the arguments.

Click here to view code image

```
List<Integer> intList1 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);   // Fixed-ar
List<Integer> intList2 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Vararg
System.out.println(intList1);        // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
System.out.println(intList2);        // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

At (1) below, an explicit array is passed as an argument, resulting in the variable arity `List.of(E...)` method being called, creating a list of `String`. At (2), the method call explicitly specifies the type of its argument as `String[]`. In this case the one-argument `List.of(E)` method is called, creating a list of length 1 and whose element type is `String[]`.

Click here to view code image

```
String[] strArray = {"Tom", "Dick", "Harriet"};
List<String> strList = List.of(strArray);                  // (1) List of Stri
List<String[]> strArrayList = List.<String[]>of(strArray); // (2) List of Stri
System.out.println(strList);       // [Tom, Dick, Harriet]
System.out.println(strArrayList);  // [[Ljava.lang.String;@3b22cdd0]
```

The code below shows how we can make a copy of a collection, in this case, a list. The `copyOf()` method creates a copy of the list passed as an argument at (1). The list created is unmodifiable analogous to the lists created with the `List.of()` methods. The code also shows that modifying the original list does *not* reflect in the copy of the list.

Click here to view code image

```
List<String> fab4 = new ArrayList<>();
fab4.add("John"); fab4.add("Paul"); fab4.add("George"); fab4.add("Ringo");
System.out.println(fab4);                     // [John, Paul, George, Ringo]
List<String> fabAlways = List.copyOf(fab4);   // (1)
fab4.remove("John"); fab4.remove("George");   // Modify original list
System.out.println(fab4);                     // [Paul, Ringo]
System.out.println(fabAlways);                // [John, Paul, George, Ringo]
```

12.3 Modifying an `ArrayList<E>`

The `ArrayList<E>` class provides methods to append, insert, replace, and remove elements from a list. In addition, it has methods to modify the capacity of a list.

Adding Elements

The various add methods allow elements to be *appended at the end of a list* and also *inserted at a specified index* in the list.

Click here to view code image

```
boolean add(E element)                                      From List<E> interface.
void add(int index, E element)                              From List<E> interface.
```

The first method will append the specified element to the *end* of the list. It returns `true` if the collection was modified as a result of the operation.

The second method inserts the specified element at the specified index. If necessary, it shifts the element previously at this index and any subsequent elements one position toward the end of the list. The method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The type parameter `E` represents the element type of the list.

Click here to view code image

```
boolean addAll(Collection<? extends E> c)                   From List<E> interface.
boolean addAll(int index,                                   From List<E> interface.
               Collection<? extends E> c)
```

The first method inserts the elements from the specified collection at the end of the list. The second method inserts the elements from the specified collection at the specified index; that is, the method splices the elements of the specified collection into the list at the specified index. These methods return `true` if any elements were added. Elements are inserted using an iterator of the specified collection (§15.2, p. 791).

The second method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The declaration of the parameter `c` essentially means that parameter `c` can refer to any collection whose element type is `E` or whose element type is a subtype of `E`.

Replacing Elements

The following methods replace elements in a list with new elements.

Click here to view code image

```
E set(int index, E element)                        From List<E> interface.
```

Replaces the element at the specified index with the specified element. It returns the previous element at the specified index. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

Click here to view code image

```
default void replaceAll(UnaryOperator<E> operator)    From List<E> interface.
```

Replaces each element of this list with the result of applying the unary operator (§13.10, p. 720) to that element. See also the `List<E>` interface (§15.3, p. 801).

Removing Elements

A summary of selected methods that can remove elements of a list is given here:

Click here to view code image

```
void clear()                                       From List<E> interface.
```

Deletes all elements from the list. The list is empty after the call, so it has size 0.

Click here to view code image

```
E remove(int index)                                From List<E> interface.
boolean remove(Object element)                     From List<E> interface.
```

The first method deletes and returns the element at the specified index. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

The second method removes the *first* occurrence of the element from the list, using object value equality. The method returns `true` if the call was successful. This method does not throw an exception if an element value is `null`, or if it is passed a `null` value.

Both methods will contract the list accordingly if any elements are removed.

Click here to view code image

```
boolean removeAll(Collection<?> c)                 From List<E> interface.
boolean removeIf(Predicate<? super E> filter)  From Collection<E> interface.
```

> The first method removes from this list all elements that are contained in the specified collection.
>
> The second method removes from this list all elements that satisfy the filtering criteria defined by a lambda expression that implements the `Predicate<T>` functional interface (§13.6, p. 703). See also filtering a list (§15.2, p. 797).
>
> Both methods return `true` if the call was successful. The list is contracted accordingly if any elements are removed.

Modifying Capacity

The following two methods can be used to modify the capacity of a list. There is no method that returns the current capacity of an `ArrayList<E>`.

> ```
> void trimToSize()
> ```
>
> Trims the capacity of this list to its current size.
>
> Click here to view code image
>
> ```
> void ensureCapacity(int minCapacity)
> ```
> From `List<E>` interface.
>
> Ensures that the capacity of this list is large enough to hold at least the number of elements specified by the minimum capacity.

All the code snippets in this section can be found in Example 12.1, p. 663. The method `printListWithIndex()` at (16) in Example 12.1 prints the elements prefixed with their index in the list, making it easier to see how the list changes structurally:

Click here to view code image

```
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

We have seen that the `add(E)` method appends an element to the end of the list. The following code adds the strings from an array of `String` to an `ArrayList` of `String`. The output from Example 12.1 at (2) shows how the elements are added at the end of the list.

Click here to view code image

```
System.out.println("\n(2) Add elements to list:");
for (String str : wordArray) {
  strList.add(str);
  printListWithIndex(strList);
}
```

We can insert a new element at a specific index using the overloaded method `add(int, E)`. The output from the following code shows how inserting an element at index 2 shifted the elements structurally in the list.

Click here to view code image

```
                              // [0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5
strList.add(2, "Java");      // Insert an element at index 2 in the list.
printListWithIndex(strList); // [0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob,
                             //  5:Rotator, 6:Bob]
```

Note that an index value equal to 0 or the size of the list is always allowed for the method `add(int, E)`.

Click here to view code image

```
List<String> list1 = new ArrayList<>();   // []
list1.add(0, "First");                    // [First]
list1.add(list1.size(), "Last");          // [First, Last]
```

We can replace an element at a specified index using the `set(int, E)` method. The method returns the element that was replaced.

Click here to view code image

```
System.out.println("(3) Replace the element at index 1:");
String oldElement = strList.set(1, "Naan");
System.out.println("Element that was replaced: " + oldElement);   // "Ada"
printListWithIndex(strList); // [0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob,
                             //  5:Rotator, 6:Bob]
```

We can remove or empty a list of all its elements using the `clear()` method:

Click here to view code image

```
// list1 is [First, Last].
list1.clear();                            // []
```

We can also remove elements from a list, with the list being contracted accordingly.

Click here to view code image

```
System.out.println("(4) Remove the element at index 0:");
System.out.println("Element removed: " + strList.remove(0));      // "level"
printListWithIndex(strList); // [0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:

System.out.println("(5) Remove the first occurrence of \"Java\":");
System.out.println("Element removed: " + strList.remove("Java")); // true
printListWithIndex(strList); // [0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]
```

The `remove(int)` removes the element at the specified index. The method `remove(Object)` needs to search the list and compare the argument object with elements in the list for object value equality. This test requires that the argument object override the `equals()` method from the `Object` class, which merely determines reference value equality. The `String` class provides the appropriate `equals()` method. However, the following code will not give the expected result because the `String-Builder` class does not provide its own `equals()` method.

Click here to view code image

```
List<StringBuilder> sbList = new ArrayList<>();
for (String str : wordArray)
  strList.add(str);
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
StringBuilder element = new StringBuilder("Ada");
System.out.println("Element to be removed: " + element);          // "Ada"
System.out.println("Element removed: " + sbList.remove(element)); // false
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
```

Once it is known that an `ArrayList<E>` will not grow in size, it might be a good idea to trim its capacity down to its size by calling the `trimToSize()` method, thereby minimizing the storage used by the `ArrayList<E>`. To reduce the number of times the capacity is increased when adding a large number of elements to the list, appropriate capacity can be set via the `ensureCapacity()` method before the operation.

Click here to view code image

```
sbList.trimToSize();           // Capacity is now same as size—that is, 6.
sbList.ensureCapacity(10000);  // Capacity is now large enough for 10000 eleme
                               // Size is still 6.
```

Primitive Values and `ArrayList`s

Since primitive values cannot be stored in an `ArrayList<E>`, we can use the wrapper classes to box such values first. In the following code, we create a list of `Integer` in which the `int` values are autoboxed in `Integer` objects and then added to the list. We try to delete the element with value `1`, but end up deleting the element at index 1 instead (i.e, the value `20`).

Click here to view code image

```
List<Integer> intList = new ArrayList<>();
intList.add(10); intList.add(20); intList.add(1);
System.out.println(intList);                                  // [10, 20, 1]
System.out.println("Element to be removed: " + 1);            // 1
System.out.println("Element removed: " + intList.remove(1));  // 20
System.out.println(intList);                                  // [10, 1]
```

The method call

```
intList.remove(1)
```

has the signature

```
intList.remove(int)
```

This signature matches the overloaded method that removes the element at a specified index, so it is this method that is called at runtime. We say that this method is the *most specific* in this case. For the code to work as intended, the primitive value must be explicitly boxed.

Click here to view code image

```
System.out.println(intList);                                  // [10, 20, 1]
System.out.println("Element to be removed: " + 1);                        // 1
System.out.println("Element removed: " +
                   intList.remove(Integer.valueOf(1)));       // true
System.out.println(intList);                                  // [10, 20]
```

The method call

Click here to view code image

```
intList.remove(Integer.valueOf(1))
```

has the signature

```
intList.remove(Integer)
```

This call matches the overloaded `remove(Object)` method, since an `Integer` object can be passed to an `Object` parameter. This method is the most specific in this case, and is executed.

12.4 Querying an `ArrayList<E>`

A summary of useful methods that can be used to query a list is provided below.

> Click here to view code image
>
> `int size()` From `List<E>` interface.
>
> Returns the number of elements currently in the list. In a non-empty list, the first element is at index 0 and the last element is at `size()-1`.
>
> Click here to view code image

```
boolean isEmpty()                               From List<E> interface.
```

Determines whether the list is empty (i.e., whether its size is 0).

Click here to view code image

```
E get(int index)                                From List<E> interface.
```

Returns the element at the specified *positional index*. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

Click here to view code image

```
boolean contains(Object element)                From List<E> interface.
```

Determines whether the argument object is contained in the collection, using object value equality. This is called the *membership test.*

Click here to view code image

```
int indexOf(Object o)                             From List<E> interface.
int lastIndexOf(Object o)                         From List<E> interface.
```

Return the indices of the first and last occurrences of the element that are equal (using object value equality) to the specified argument, respectively, if such an element exists in the list; otherwise, the value `-1` is returned. These methods provide *element search* in the list.

Click here to view code image

```
List<E> subList(int fromIndex, int toIndex)     From List<E> interface.
```

Returns a *view* of the list, which consists of the sublist of the elements from the index `fromIndex` to the index `toIndex-1` (i.e., a half-open interval). A view allows the range it represents in the underlying list to be manipulated. Any changes in the view are reflected in the underlying list, and vice versa. Views can be used to perform operations on specific ranges of a list.

Click here to view code image

```
boolean equals(Object o)                        From List<E> interface.
```

Compares the specified object with this list for object value equality. It returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal according to object value equality.

The method `size()` returns the number of elements in the list, and the method `empty()` determines whether the list is empty.

Click here to view code image

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("The size of the list is " + strList.size());          // 5
boolean result = strList.isEmpty();
System.out.println("The list " + (result ? "is" : "is not") + " empty."); // f
```

The method `get(int)` retrieves the element at the specified index.

Click here to view code image

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("First element: " + strList.get(0);                 // Naan
System.out.println("Last element: " + strList.get(strList.size()-1));  // Bob
```

The `equals()` method of the `ArrayList` class can be used to compare two lists for equality with regard to size and corresponding elements being equal in each list.

Click here to view code image

```
List<String> strList2 = new ArrayList<>(strList);
boolean trueOrFalse = strList.equals(strList2);                        // true
```

The method `subList()` returns a *view* of a list—that is, a sublist of the list. As the view is backed by the underlying list, operations on the sublist will be reflected in the underlying list, as demonstrated by the following code:

Click here to view code image

```
out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator, Bob
List<String> strList3 = strList.subList(1, 4);
out.println("Sublist before remove: " + strList3);     // [kayak, Bob, Rotator]
out.println("Remove: " + strList3.get(0));  // "kayak"
strList3.remove(0);                         // Remove element at index 0
out.println("Sublist after remove: " + strList3);      // [Bob, Rotator]
out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]
```

The membership test is carried out by the `contains(Object)` method. We can find the index of a specified element in the list by using the `indexOf()` and `lastIndexOf()` methods.

Click here to view code image

```
boolean found = strList.contains("Naan");     // true
int index = strList.indexOf("Bob");           // 2
index = strList.indexOf("BOB");               // -1 (Not found)
index = strList.lastIndexOf("Bob");           // 4 (Last occurrence)
```

Again, these methods require that the element type provide a meaningful `equals()` method for object value equality testing.

12.5 Iterating Over an `ArrayList<E>`

Various methods for iterating over collections are discussed in §15.2, p. 791. Here we look at a very common task of iterating over a list to perform some operation on each element of the list.

We can use positional access to iterate over a list with the `for(;;)` loop. The generic method `printListWithIndex()` in Example 12.1 uses the `for(;;)` loop to create a new `ArrayList` of `String` that contains each element of the argument list prefixed with the index of the element.

Click here to view code image

```
public static <E> void printListWithIndex(List<E> list) {
  List<String> newList = new ArrayList<>();
  for (int i = 0; i < list.size(); i++) {
    newList.add(i + ":" + list.get(i));
  }
  System.out.println(newList);
}
```

Sample output from the method call `printListWithIndex(strList)` is shown here:

Click here to view code image

```
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
```

The method `printListWithIndex()` in Example 12.1 can print *any* list in this format. Its header declaration says that it accepts a list of element type `E`. The element type `E` is determined from the method call. In the preceding example, `E` is determined to be `String`, as a `List` of `String` is passed in the method call.

Since the `ArrayList<E>` class implements the `Iterable<E>` interface (i.e., the class provides an iterator), we can use the `for(:)` loop to iterate over a list.

```
for (String str : strList) {
  System.out.print(str + " ");
}
```

The `ArrayList<E>` also provides specialized iterators to iterate over a list (§15.3, p. 801).

For performing a given action on each element of a list, the `forEach()` method can be used (§15.2, p. 796), where the action is specified by a consumer (§13.7, p. 709).

One pertinent question to ask is how to remove elements from the list when iterating over the list. The `for(:)` loop does not allow the list structure to be modified:

Click here to view code image

```
for (String str : strList) {
  if (str.length() <= 3) {
    strList.remove(str);                // Throws ConcurrentModificationExcepti
  }
}
```

We can use positional access in a loop to iterate over the list, but we must be careful in updating the loop variable, as the list contracts when an element is removed. Better solutions for this purpose are discussed in §15.2, p. 796.

12.6 Converting an `ArrayList<E>` to an Array

The following methods are specified by the `Collection<E>` interface and can be used to convert a collection to an array. List and set implementations in the `java.util` package provide customized versions of the first two methods for this purpose. In this section we consider how to convert lists to arrays.

Click here to view code image

```
Object[] toArray()                                From Collection<E> interface.
<T> T[] toArray(T[] a)                            From Collection<E> interface.
```

The first method returns an array of type `Object` filled with all the elements of a collection. The returned array can be modified independently of the list from which it was created.

The second method is a generic method that stores the elements of a collection in an array of type `T`. If the specified array is big enough, the elements are stored in this array. If there is room to spare in the array—that is, if the length of the array is greater than the number of elements in the collection—the element found immediately after storing the elements of the collection is set to the `null` value before the array is returned. If the array is too small, a new array of type `T` and appropriate size is created. If `T` is not a supertype of the runtime type of every element in the collection, an

`ArrayStoreException` is thrown.

Click here to view code image

```
default <T> T[]
        toArray(IntFunction<T[]> generator)     From Collection<E> interface.
```

Allows creation of an array of a particular runtime type given by the parameterization of the type parameter `T[]`, using the specified `generator` function (§13.8, p. 717) to allocate the array of the desired type and the specified length.

The `default` implementation calls the `generator` function with 0 and then passes the resulting array of length 0 to the `toArray(T[])` generic method.

See also array operations in the `Collection<E>` interface (§15.2, p. 798).

The actual element type of the elements in the `Object` array returned by the first `toArray()` method can be any subtype of `Object`. It may be necessary to cast the `Object` reference of an element to the appropriate type, as in the following code:

Click here to view code image

```
System.out.println("(15) Convert list to array:");
Object[] objArray = strList.toArray();                     // Object[]
System.out.println("Object[] length: " + objArray.length); // 5
System.out.print("Length of each string in the Object array: ");
for (Object obj : objArray) {
  String str = (String) obj;                               // Cast required.

  System.out.print(str.length() + " ");
}
System.out.println();
```

The generic `toArray()` method returns an array of type `T`, when it is passed an array of type `T` as an argument. In the following code, the array of `String` that is returned has the same length as the size of the list of `String`, even though a `String` array of length 0 was passed as an argument:

Click here to view code image

```
String[] strArray = strList.toArray(new String[0]);         // String[]
System.out.println("String[] length: " + strArray.length);  // 5
System.out.print("Length of each string in the String array: ");
for (String str : strArray) {
  System.out.print(str.length() + " ");
}
System.out.println();
```

12.7 Creating List Views

The `asList()` method in the `Arrays` class and the `toArray()` methods in the `Collection<E>` interface provide the bidirectional bridge between arrays and collections. The `asList()` method of the `Arrays` class creates `List<E>` views of arrays.

> Click here to view code image
>
> ```
> @SafeVarargs <E> List<E> asList(E... elements)                    From Arrays
> ```
>
> Returns a *fixed-size list view* that is backed by the *array* corresponding to the variable arity parameter `elements`. The method is annotated with `@SafeVarargs` because of the variable arity parameter. The annotation suppresses the heap pollution warning in its declaration and also unchecked generic array creation warning at the call sites (§25.5, p. 1585).

Changes to the elements of the list view are reflected in the array, and vice versa. The list view is said to be *backed* by the array. The size of the list view is equal to the array length and *cannot* be changed. The iterator for a list view does not support the `remove()` method.

The code below illustrates use of the `asList()` method. The `list1` at (1) is backed by the `array1`. The `list2` is backed by an implicit array of `Integer` at (2). An array of a primitive type cannot be passed as an argument to this method, as evident by the compile-time error at (3). However, the `Collections.addAll()` method provides better performance when adding a few elements to an *existing* collection.

Click here to view code image

```
Integer[] array1 = new Integer[] {9, 1, 1};
List<Integer> list1 = Arrays.asList(array1);         // (1) A list of Integer
List<Integer> list2 = Arrays.asList(9, 1, 1);        // (2) Varargs

int[] array2 = new int[] {9, 1, 1};                  // An array of int
// List<Integer> intList3 = Arrays.asList(array2);   // (3) Compile-time erro
```

Various operations on the `list1` show how changes are reflected in the backing `array1`. Elements cannot be added to the list view (shown at (4)), and elements cannot be removed from the list view (shown at (9)). An `UnsupportedOperationException` is thrown in both cases. An element at a given position can be changed, as shown at (5). The change is reflected in the `list1` and the `array1`, as shown at (6) and (7), respectively. A sublist view is created from the `list1` at (8), and sorted at (10). The changes in the `sublist1` are reflected in the `list1` and the backing `array1`.

Click here to view code image

```
System.out.println(list1);                   // [9, 1, 1]
// list1.add(10);                            // (4) UnsupportedOperationExcept
list1.set(0, 10);                            // (5)
System.out.println(list1);                   // (6) [10, 1, 1]
System.out.println(Arrays.toString(array1)); // (7) [10, 1, 1]
List<Integer> sublist1 = list1.subList(0, 2);// (8)
System.out.println(sublist1);                // [10, 1]
// sublist1.clear();                         // (9) UnsupportedOperationExcept
Collections.sort(sublist1);                  // (10)
System.out.println(sublist1);                // [1, 10]
System.out.println(list1);                   // [1, 10, 1]
System.out.println(Arrays.toString(array1)); // [1, 10, 1]
```

The code below shows how duplicates can be eliminated from an array:

Click here to view code image

```
String[] jiveArray   = new String[] {"java", "jive", "java", "jive"};
Set<String> jiveSet  = new HashSet<>(Arrays.asList(jiveArray));// (1)
String[] uniqueJiveArray = jiveSet.toArray(new String[0]);     // (2)
System.out.println(Arrays.toString(uniqueJiveArray));           // (3) [java,
```

At (1), the `jiveArray` is used to create a `List`, which in turn is used to create a `Set`. At (2), the argument to the `toArray()` method specifies the type of the array to be created from the set. The final array `uniqueJiveArray` does not contain duplicates, as can be seen at (3).

Comparing Unmodifiable Lists and List Views

There are subtle differences to be aware of between unmodifiable lists and list views.

- *Backing an array*

  The `Arrays.asList()` method returns a *fixed-size list view* that is backed by the array passed as an argument so that any changes made to the array are reflected in the view list as well. This is not true of the `List.of()` and `List.ofCopy()` methods, as they create *unmodifiable lists* which are *not backed* by any argument array that is passed either explicitly or implicitly as a variable arity parameter. In the code below, we see that the list view returned by the

`Arrays.asList()` method reflects the change at (1) in its backing array, but not the unmodifiable list returned by the `List.of()` method at (2) when its argument array is modified.

Click here to view code image

```
Integer[] yrArray1 = {2020, 2021, 2022};
List<Integer> yrlist1 = Arrays.asList(yrArray1);
yrArray1[0] = 2019;                                    // Modify the a
out.println("yrArray1: " + Arrays.toString(yrArray1)); //     [2019, 2
out.println("yrlist1: " + yrlist1);                    // (1) [2019, 2

Integer[] yrArray2 = {2020, 2021, 2022};
List<Integer> yrlist2 = List.of(yrArray2);
yrArray2[0] = 2019;                                    // Modify the a
out.println("yrArray2: " + Arrays.toString(yrArray2)); //     [2019, 2
out.println("yrlist2: " + yrlist2);                    // (2) [2020, 2
```

- *Mutability*

  The list view returned by the `Arrays.asList()` method is *mutable*, but it cannot be structurally modified. In contrast, the unmodifiable list returned by the `List.of()` method is *immutable*.
  In the code below, only the list view returned by the `Arrays.asList()` method can be modified as shown at (1), but an attempt to modify the unmodifiable list returned by the `List.of()` method at (2) throws an exception.

  Click here to view code image

```
List<Integer> yrList3 = Arrays.asList(2020, 2021, 2022);
yrList3.set(2, 2023);                     // (1) OK
out.println(yrList3);                     // [2020, 2021, 2023]

List<Integer> yrlist4 = List.of(2020, 2021, 2022);
yrlist4.set(2, 2023);                     // (2) UnsupportedOperationE
```

  However, both lists will throw an exception if an attempt is made to change them *structurally*—that is, add or remove elements from the list:

  Click here to view code image

```
yrList3.add(2050);                        // UnsupportedOperationExcep
yrlist4.remove(0);                        // UnsupportedOperationExcep
```

- *The* `null` *value*

  The `Arrays.asList()` method allows `null` elements, whereas the `List.of()` and `List.ofCopy()` methods do not.

Click here to view code image

```
List<Integer> yrList5 = Arrays.asList(2020, 2021, null);  // OK.
List<Integer> yrlist6 = List.of(2020, 2021, null);        // NullPoint
```

The behavior of the `List.contains()` method when passed the `null` value is dependent on which method created the list.

Click here to view code image

```
boolean flag1 = Arrays.asList(2021, 2022).contains(null); // OK.
boolean flag2 = List.of(2021, 2022).contains(null);       // NullPoint
```

12.8 Arrays versus `ArrayList`s

Table 12.1 summarizes the differences between arrays and `ArrayList`s.

Table 12.1 *Summary of Arrays versus* `ArrayList`s

	Arrays	**`ArrayList`**
Construct support	Built into the language.	Provided by the generic class `ArrayList<E>`.
Initial length/size specification	Length is specified in the array construction expression, directly or indirectly by the initialization block.	Cannot specify size at construction time. However, initial capacity can be specified.
Length/size	Length of an array is static (fixed) once it is created. Each array has a `public final int` field called `length`. (The `String` and the `StringBuilder` class provide the method `length()` for this purpose.)	Size can change dynamically. `ArrayList<E>` provides the method `size()` to obtain the current size of the list.
Element type	Primitive and reference types.	Only reference types.
Operations on elements	An element in the array is designated by the array name and an index using the `[]` operator, and can be used as a simple variable.	The `ArrayList<E>` class provides various methods to add, insert, replace, retrieve, and remove elements from a list.
Iterator	Arrays do not provide iterators, apart from using the `for(:)` loop for traversal.	The `ArrayList<E>` class provides customized iterators for lists, in addition to the `for(:)` loop for iterating over the elements (§15.2, p. 791).
Generics	Cannot create arrays of generic types using the `new` operator. Runtime check required for storage at runtime.	`ArrayList<E>` is a generic type. Can create parameterized `ArrayList`s of reference types using the `new` operator. No runtime check required for storage at runtime, as type-safety is checked at compile time.
Subtype relationship	Arrays are covariant: subtype relationship between their element types implies subtype relationship between the arrays.	Subtype relationship between two reference types does not imply subtype relationship between `ArrayList`s of the two types—that is, there is no subtype relationship.
Sorting	`java.util.Arrays.sort(array)` `java.util.Arrays.sort(array, comparator)`	`java.util.Collections.sort(list)` `java.util.Collections.sort(list, comparator)`

(§15.12, p. 864)

`java.util.List.sort(comparator)`

(§15.11, p. 856)

Text representation

`java.util.Arrays.toString(array)`

Example 12.1 is a collection of code snippets used throughout this chapter to illustrate the various methods of the `ArrayList<E>` class.

Example 12.1 *Using an* `ArrayList`

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class ArrayListMethods {

  public static void main(String[] args) {

    String[] wordArray = { "level", "Ada", "kayak", "Bob", "Rotator", "Bob" };

    out.println("(1) Create an empty list of strings:");
    List<String> strList = new ArrayList<>();
    printListWithIndex(strList);

    out.println("\n(2) Add elements to list:");
    for (String str : wordArray) {
      strList.add(str);
      printListWithIndex(strList);
    }
    out.println("Insert an element at index 2 in the list:");
    strList.add(2, "Java");
    printListWithIndex(strList);

    out.println("\n(3) Replace the element at index 1:");
    String oldElement = strList.set(1, "Naan");
    out.println("Element that was replaced: " + oldElement);
    printListWithIndex(strList);

    out.println("\n(4) Remove the element at index 0:");
    out.println("Element removed: " + strList.remove(0));
    printListWithIndex(strList);

    out.println("\n(5) Remove the first occurrence of \"Java\":");
    out.println("Element removed: " + strList.remove("Java"));
    printListWithIndex(strList);

    out.println("\n(6) Determine the size of the list:");
    out.println("The size of the list is " + strList.size());

    out.println("\n(7) Determine if the list is empty:");
    boolean result = strList.isEmpty();
    out.println("The list " + (result ? "is" : "is not") + " empty.");

    out.println("\n(8) Get the element at specific index:");
    out.println("First element: " + strList.get(0));
    out.println("Last element: " + strList.get(strList.size() - 1));
    out.println("\n(9) Compare two lists:");
    List<String> strList2 = new ArrayList<>(strList);
    boolean trueOrFalse = strList.equals(strList2);
    out.println("The lists strList and strList2 are"
        + (trueOrFalse ? "" : " not") + " equal.");
    strList2.add(null);
    printListWithIndex(strList2);
    trueOrFalse = strList.equals(strList2);
    out.println("The lists strList and strList2 are"
        + (trueOrFalse ? "" : " not") + " equal.");

    out.println("\n(10) Sublists as views:");
    out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator,
    List<String> strList3 = strList.subList(1, 4);
    out.println("Sublist before remove: " + strList3);    // [kayak, Bob, Rota
    out.println("Remove: " + strList3.get(0));  // "kayak"
    strList3.remove(0);                         // Remove element at index 0
    out.println("Sublist after remove: " + strList3);     // [Bob, Rotator]
    out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]

    out.println("\n(11) Membership test:");
    boolean found = strList.contains("Naan");
    String msg = found ? "contains" : "does not contain";
    out.println("The list " + msg + " the string \"Naan\".");

    out.println("\n(12) Find the index of an element:");
    int pos = strList.indexOf("Bob");
    out.println("The index of string \"Bob\" is: " + pos);
    pos = strList.indexOf("BOB");
    out.println("The index of string \"BOB\" is: " + pos);
    pos = strList.lastIndexOf("Bob");
    out.println("The last index of string \"Bob\" is: " + pos);
    printListWithIndex(strList);

    out.println("\n(13) Iterating over the list using the for(;;) loop:");
    for (int i = 0; i < strList.size(); i++) {
      out.print(i + ":" + strList.get(i) + " ");
    }
    out.println();

    out.println("\n(14) Iterating over the list using the for(:) loop:");
    for (String str : strList) {
      out.print(str +  " ");
      // strList.remove(str);        // Throws ConcurrentModificationException
    }
    out.println();

    out.println("\n(15) Convert list to array:");
    Object[] objArray = strList.toArray();
    out.println("Object[] length: " + objArray.length);
    out.print("Length of each string in the Object array: ");
    for (Object obj : objArray) {
      String str = (String) obj; // Cast required.
      out.print(str.length() + " ");
    }
    out.println();
    String[] strArray = strList.toArray(new String[0]);
    out.println("String[] length: " + strArray.length);
    out.print("Length of each string in the String array: ");
    for (String str : strArray) {
      out.print(str.length() + " ");
    }
  }
  /**
   * Print the elements of a list, together with their index:
   * [0:value0, 1:value1, ...]
   * @param list   List to print with index
   */
  public static <E> void printListWithIndex(List<E> list) {             // (16)
    List<String> newList = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
      newList.add(i + ":" + list.get(i));
    }
    out.println(newList);
  }
}
```

Output from the program:

Click here to view code image

```
(1) Create an empty list of strings:
[]

(2) Add elements to list:
[0:level]
[0:level, 1:Ada]
[0:level, 1:Ada, 2:kayak]
[0:level, 1:Ada, 2:kayak, 3:Bob]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
Insert an element at index 2 in the list:
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]

(3) Replace the element at index 1:
Element that was replaced: Ada
[0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]

(4) Remove the element at index 0:
Element removed: level
[0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

(5) Remove the first occurrence of "Java":
Element removed: true
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]

(6) Determine the size of the list:
The size of the list is 5

(7) Determine if the list is empty:
The list is not empty.

(8) Get the element at specific index:
First element: Naan
Last element: Bob

(9) Compare two lists:
The lists strList and strList2 are equal.
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob, 5:null]
The lists strList and strList2 are not equal.

(10) Sublists as views:
Underlying list: [Naan, kayak, Bob, Rotator, Bob]
Sublist before remove: [kayak, Bob, Rotator]
Remove: kayak
Sublist after remove: [Bob, Rotator]
Underlying list: [Naan, Bob, Rotator, Bob]

(11) Membership test:
The list contains the string "Naan".

(12) Find the index of an element:
The index of string "Bob" is: 1
The index of string "BOB" is: -1
The last index of string "Bob" is: 3
[0:Naan, 1:Bob, 2:Rotator, 3:Bob]

(13) Iterating over the list using the for(;;) loop:
0:Naan 1:Bob 2:Rotator 3:Bob

(14) Iterating over the list using the for(:) loop:
Naan Bob Rotator Bob

(15) Convert list to array:
Object[] length: 4
Length of each string in the Object array: 4 3 7 3
String[] length: 4
Length of each string in the String array: 4 3 7 3
```

Review Questions

12.1 Which statement is true about the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A10 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    for (int i = 0; i < strList.size(); /* empty */) {
      if (strList.get(i).length() <= 3) {
        strList.remove(i);
      } else {
        ++i;
      }
    }
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `ConcurrentModificationException` at runtime.

d. The program will not terminate when run.

e. The program will print `[Anna, Adda]`.

f. The program will print `[Anna, Ada, Bob, Adda]`.

12.2 Which of the following statements are true about the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A15 {
  public static void main(String[] args) {
    doIt1(); doIt2();
  }

  public static void doIt1() {
    List<StringBuilder> sbListOne = new ArrayList<>();
    sbListOne.add(new StringBuilder("Anna"));
    sbListOne.add(new StringBuilder("Ada"));
    sbListOne.add(new StringBuilder("Bob"));
    List<StringBuilder> sbListTwo = new ArrayList<>(sbListOne);
    sbListOne.add(null);
    sbListTwo.get(1).reverse();
    System.out.println(sbListOne);                                  // (
  }

  public static void doIt2() {
    List<String> listOne = new ArrayList<>();
    listOne.add("Anna"); listOne.add("Ada"); listOne.add("Bob");
    List<String> listTwo = new ArrayList<>(listOne);
    String strTemp = listOne.get(0);
    listOne.set(0, listOne.get(listOne.size()-1));
    listOne.set(listOne.size()-1, strTemp);
    System.out.println(listTwo);                                    // (
  }
}
```

Select the two correct answers.

a. (1) will print `[Anna, Ada, Bob, null]`.

b. (1) will print `[Anna, adA, Bob, null]`.

c. (2) will print `[Anna, Ada, Bob]`.

d. (2) will print `[Bob, Ada, Anna]`.

e. The program will throw an `IndexOutOfBoundsException` at runtime.

12.3 Which statement is true about the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A20 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        System.out.print(i);
      }
    }
    System.out.println();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `NullPointerException` at runtime.

d. The program will print `34`.

12.4 Which statement is true about the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A30 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        strList.remove(i);
      }
    }
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `NullPointerException` at runtime.

d. The program will throw a `ConcurrentModificationException` at runtime.

e. The program will not terminate when run.

f. The program will print `[Anna, Ada, Bob]`.

g. The program will print `[Anna, Ada]`.

12.5 Which statement is true about the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A40 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    while (strList.remove("Bob"));
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw a `NullPointerException` at runtime.

c. The program will not terminate when run.

d. The program will print `[Anna, Ada, Adda]`.

e. The program will print `[Anna, Ada, Bob, Adda]`.

f. The program will print `[Anna, Ada, null, Adda]`.

g. The program will print `[Anna, Ada, null, Bob, Adda]`.

12.6 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_01 {
  public static void main(String[] args) {

    String[] data1 = {"A","B","B","A"};
    List<String> data2 = new ArrayList<>();
    for (String s : data1) {
      data2.add(s);
    }
    data2.set(1, "X");
    data2.add(1, "X");
    data2.remove(2);
    System.out.println(data2);
  }
}
```

Select the one correct answer.

a. `[X, B, B, A]`

b. `[A, X, B, A]`

c. `[A, X, A]`

d. `[A, X, X]`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.7 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_02 {
  public static void main(String[] args) {
    String[] data1 = {"A","B","B","A"};
    List<String> data2 = Arrays.asList(data1);
    data2.set(1, "X");
    data2.set(2, "X");
    System.out.println(data2);
  }
}
```

Select the one correct answer.

a. `[A, X, X, A]`

b. `[X, X, B, A]`

c. `[A, X, X, B, B, A]`

d. `[X, X, A, B, B, A]`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.8 What will be the result of running the following program?

Click here to view code image

```
public class Song {
  private String name;
  public Song(String name) {
    this.name = name;
  }
  public void update() {
    name = name.toUpperCase();
  }
  public String toString() {
    return name;
  }
}
```

Click here to view code image

```
import java.util.*;
public class Test12_04 {
  public static void main(String[] args) {
    Song[] playArray1 = {new Song("a"), new Song("b")};
    List<Song> playlist = Arrays.asList(playArray1);
    Song[] playArray2 = playlist.toArray(new Song[]{});
    playArray1[1].update();
    System.out.print(playArray1[1]);
    System.out.print(playlist.get(1));
    System.out.print(playArray2[1]);
  }
}
```

Select the one correct answer.

a. `Bbb`

b. `BBb`

c. `BBB`

d. `bbb`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.9 What will be the result of running the following program?

Click here to view code image

```
public class MySong {
  private String name;
  public MySong(String name) {
    this.name = name;
  }
  public String toString() {
    return name;
  }
}
```

Click here to view code image

```
import java.util.*;
public class Test12_05 {
  public static void main(String[] args) {
    MySong[] playArray1 = {new MySong("A"), new MySong("B")};
    List<MySong> playlist = List.of(playArray1);
    MySong[] playlist2 = playlist.toArray(new MySong[]{});
    playArray1[0] = new MySong("C");
    System.out.print(playArray1[0]);
    System.out.print(playlist.get(0));
    System.out.print(playlist2[0]);
  }
}
```

Select the one correct answer.

a. `CCA`

b. `CAA`

c. `CCC`

d. `AAA`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.10 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_06 {
  public static void main(String[] args) {
    List<String> data1 = List.of("A","B","C");
    String[] data2 = data1.toArray(new String[]{"X","Y","Z"});
    data2[1] = data1.get(0).toLowerCase();
    for (String s: data2) {
      System.out.print(s);
    }
  }
}
```

Select the one correct answer.

a. `AaC`

b. `AbC`

c. `XaZ`

d. `XbZ`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.11 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_07 {
  public static void main(String[] args) {

    List<Character> text = new ArrayList<>(3);
    for (char a = 'a'; a <= 'e'; a++) {
      text.add(a);
    }
    System.out.println(text);
  }
}
```

Select the one correct answer.

- **a.** `[a, b, c]`
- **b.** `[a, b, c, d, e]`
- **c.** `[a, b, c, d]`
- **d.** The program will throw an exception at runtime.
- **e.** The program will fail to compile.

Functional-Style Programming 13

Chapter Topics

- Declaring functional interfaces
- Defining lambda expressions
- Implementing functional interfaces using lambda expressions
- Defining and type checking lambda expressions in the context of a functional interface
- Understanding deferred execution of a lambda expression
- Understanding the implications of using class members from the enclosing class, and of using local variables from the enclosing method in a lambda expression
- Comparing lambda expressions and anonymous classes
- Understanding behavior parameterization in functional-style programing
- Using built-in functional interfaces in the `java.util.function` package: suppliers, predicates, consumers, and functions
- Composing compound operators with built-in functional interfaces
- Using specializations of built-in functional interfaces
- Using primitive type specializations of built-in functional interfaces
- Using method and constructor references

- Understanding the contexts in which lambda expressions can be defined

Java SE 17 Developer Exam O

[6.1] Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to supply, filter, map, consume, and sort data

§13.1, p. 675 to §13.14, p. 733

- *Lambda expressions to implement functional interfaces are covered in this chapter.*
- *For streams, see Chapter 16, p. 879.*

Java SE 11 Developer Exam O

[6.1] Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package

§13.1, p. 675 to §13.14, p. 733

In many ways, Java 8 represented a watershed in the history of the language. Before Java 8, the language supported only object-oriented programming. Packing state and behavior into objects that communicate in a procedural manner was the order of the day. Java 8 brought *functional-style programming* into the language, where code representing *functionality* could be passed as values to tailor the *behavior* of methods.

We first look at the two language features that are the basis for functional-style programming in Java—functional interfaces and lambda expressions—before diving into a comparison of lambda expressions and anonymous classes, method and constructor references, and coverage of the built-in general-purpose generic functional interfaces provided by the `java.util.function` package. Subsequent chapters provide ample examples of this functional-style programming paradigm.

13.1 Functional Interfaces

Functional interfaces and lambda expressions together facilitate *behavior parameterization* (p. 691), a powerful programming paradigm that allows code representing behavior to be passed around as values, and executed when the abstract method of the functional interface is invoked. This approach is scalable, requiring only a lambda expression to implement the functional interface.

Declaring a Functional Interface

A functional interface has exactly one abstract method. This abstract method is called the *functional method* of that interface. Like any other interface, a functional interface can declare other interface members, including `static`, `default`, and `private` methods (§5.6, p. 237). Here we concentrate on the single abstract method of a functional interface. This single abstract method, like all abstract methods in an interface, is implicitly `abstract` and `public`.

The `StrPredicate` interface shown below has only one abstract method and is, by definition, a functional interface.

Click here to view code image

```
@FunctionalInterface                             // Annotation.
interface StrPredicate {
  boolean test(String str);                      // Sole public abstract metho
}
```

The annotation `@FunctionalInterface` (§25.5, p. 1579) is optional when defining functional interfaces. If the annotation is specified, the compiler will issue an error if the declaration violates the definition of a functional interface, and the Javadoc tool will also automatically generate an explanation about the functional nature of the interface. In the absence of this annotation, there is no clue from the compiler to assert that an interface is a functional interface.

The `CharSeqPredicate` interface declaration below has two abstract methods, albeit they are overloaded, but it is not a functional interface.

Click here to view code image

```
@FunctionalInterface
interface CharSeqPredicate {                              // Compile-time erro
  boolean test(String str);                               // Abstract method.
  boolean test(StringBuilder sb);                         // Abstract method.
}
```

The functional interface `NewStrPredicate` below declares only one abstract method at (1). In addition, it provides the implementations of one `default` and one `static` method at (2) and (3), respectively.

Click here to view code image

```
@FunctionalInterface
interface NewStrPredicate {
  boolean test(String str);                                 // (1) Abstract me
  default void msg(String str) { System.out.println(str); } // (2) Default met
  static void info() { System.out.println("Testing!"); }    // (3) Static meth
}
```

An interface can provide *explicit* `public abstract` method declarations for any of the three *non*-`final` `public` instance methods in the `Object` class (`equals()`, `hashCode()`, `toString()`). Including these methods explicitly in an interface should not be attempted, unless there are compelling reasons for doing so. These methods are automatically implemented by every class that implements an interface, since every class directly or indirectly inherits from the `Object` class. Therefore, such method declarations are not considered abstract in the definition of a functional interface.

In the `Funky` interface below, the first three abstract method declarations are those of the non-`final` `public` methods from the `Object` class. As explained above, these methods do not count toward the definition of a functional interface. Effectively, the `Funky` interface declares only one abstract method given by the last abstract method declaration.

Click here to view code image

```
@FunctionalInterface
interface Funky {
  @Override int hashCode();                                 // From Object class
  @Override boolean equals(Object obj);                     // From Object class
  @Override String toString();                              // From Object class
  boolean doTheFunk(Object obj);                            // Abstract method
}
```

The Java SE 17 platform API provides general-purpose functional interfaces in the `java.util.function` package that are discussed later (p. 695). Partial declaration of one such functional interface is shown below. It is the generic version of the `StrPredicate` functional interface defined earlier. The `test()` method of the `Predicate<T>` functional interface defines a `boolean`-valued predicate and evaluates it on the given object.

Click here to view code image

```
@FunctionalInterface
interface Predicate<T> {
  boolean test(T element);                                  // Functional method

  // ...
}
```

Just like any other interface, a functional interface can be implemented by a class. The following generic method takes an object of type `T` and a `Predicate<T>` and determines whether the object satisfies the predicate.

Click here to view code image

```
public static <T> boolean testPredicate(T object, Predicate<T> predicate) {
  return predicate.test(object);
}
```

Below are two implementations of the parameterized functional interface `Predicate<String>` using a concrete class and an anonymous class, respectively.

Click here to view code image

```
// Class implementation of Predicate<String>.
class PredicateTest implements Predicate<String> {
  public boolean test(String str) {
    return str.startsWith("A");                // (1)
  }
}

// An anonymous class implementation of Predicate<String>.
static Predicate<String> testLength = new Predicate<>() {
  public boolean test(String str) {
    return str.length() < 4;                   // (2)
  }
};

// Client code:
System.out.println(testPredicate("Anna", new PredicateTest()));   // true
System.out.println(testPredicate("Anna", testLength));            // false
```

Note that for any combination of an object type and a predicate criteria, the `Predicate<T>` interface has to be implemented by a *new* class (or a *new* anonymous class) in order to utilize the `testPredicate()` method. Looking at the two implementations of the `Predicate<String>` above, it is essentially the expressions in the respective `return` statements at (1) and (2) that are different. This is where lambda expressions come in, allowing more concise implementation of the abstract method of a functional interface, without requiring all the boilerplate code.

Overriding Abstract Methods from Multiple Interfaces, Revisited

A general discussion on overriding abstract methods from multiple superinterfaces can be found in §11.12, p. 621. In general, an interface can inherit multiple abstract methods from its superinterfaces, but a functional interface can only have a single abstract method. Note that superinterfaces need not be functional interfaces. It is even more imperative to use the `@FunctionalInterface` annotation to catch any inadvertent errors when a functional interface inherits abstract methods from multiple superinterfaces.

In the code below, the compiler does not flag any error in the declaration of the sub-interface `ContractZ`. It is perfectly valid for the subinterface to inherit the `doIt()` method from each of its superinterfaces, as the two method declarations will be overloaded.

Click here to view code image

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
interface ContractZ extends ContractX, ContractY {
  @Override void doIt(int d);                              // Can be omitted.
  @Override void doIt(double d);                           // Can be omitted.
}
```

Trying to declare the subinterface `ContractZ` as a functional interface, however, does not work. The multiple abstract methods inherited by the `ContractZ` are not override-equivalent—their signatures are different—and there is no single method that can override these methods, thus disqualifying `ContractZ` as a functional interface.

Click here to view code image

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
@FunctionalInterface interface ContractZ                   // Compile-time err
                     extends ContractX, ContractY {

  @Override void doIt(int d);                              // Compile-time err
  @Override void doIt(double d);                           // Compile-time err
}
```

In the code below, the subinterface `Features<T>` at (1) is a functional interface, as the declarations of the abstract method `flatten()` in its superinterfaces are override-equivalent. Any declaration of the abstract method `flatten()` can represent the abstract methods from the superinterfaces in the subinterface.

Click here to view code image

```
interface Feature1<R> { void flatten(List<R> plist); }
interface Feature2<S> { void flatten(List<S> plist); }
@FunctionalInterface
interface Features<T> extends Feature1<T>, Feature2<T> {   // (1)
  @Override void flatten(List<T> plist);                   // Can be omitted.
}
```

Selected Interfaces in the Java SE Platform API

A functional interface, like any other interface, can always be implemented by a class. The `@FunctionalInterface` annotation on a functional interface in the Java SE Platform API documentation indicates that the implementation of the functional interface is meant to be implemented by lambda expressions.

In the Java SE Platform API, not all interfaces with one abstract method declaration are marked with the `@FunctionalInterface` annotation. Such single-abstract-method (*SAM*) interfaces usually elicit specific behavior in objects which is best provided by the class of an object, and not by users of the class. Examples of single-abstract-method interfaces that should be implemented by classes include the following interfaces from the `java.lang` package:

- `Comparable<T>` for creating an object that can be compared with another object according to their natural ordering (§14.4, p. 761)
- `Iterable<T>` for creating an object that represents a collection in a `for(:)` loop to iterate over its elements (§15.2, p. 791)
- `AutoCloseable` for creating an object that represents a resource that is automatically closed when exiting a `try`-with-resources statement (§7.7, p. 412)
- `Readable` for creating an object that is a source for reading characters (§20.3, p. 1249)

The Java SE Platform API also has ample examples of functional interfaces that are intended to be implemented by lambda expressions. Built-in general-purpose functional interfaces provided in the `java.util.function` package are discussed in §13.4, p. 695. In addition, the following specialized functional interfaces are worth noting:

- `java.util.Comparator<T>` with the method `compare()` for defining criteria for comparing two objects according to their total ordering (§14.5, p. 769)
- `java.lang.Runnable` with the method `run()` for defining code to be executed in threads (§22.3, p. 1370)
- `java.util.concurrent.Callable<V>` with the method `call()` for defining code to be executed in threads, that can return a result and throw an exception (§23.2, p. 1423)

13.2 Lambda Expressions

Lambda expressions *implement* functional interfaces, and thereby define *anonymous functions*—that is, functions that do not have a name. They can be used as a *value* in a program, without the excess baggage of first being packaged into objects. As we shall see, these two features together facilitate *behavior parameterization*— that is, customizing method behavior by passing code as values.

A lambda expression has the following syntax:

formal_parameter_list `->` *lambda_body*

The parameter list and the body are separated by the `->` operator (a.k.a. the *function arrow*). The arrow operator has the penultimate level of precedence, only higher than the assignment operators which have the lowest precedence, and is right-associative.

The lambda expression syntax resembles a simplified declaration of a method, without many of the bells and whistles of a method declaration—omitting, in particular, any modifiers, return type, or `throws` clause specification. That is important, as it avoids verbosity and provides a simple and succinct notation to write lambda expressions on the fly.

A lambda expression can only occur in specific contexts: for example, as the value on the right-hand side of an assignment, as an argument passed in a method or constructor call, or as the value to cast with the cast operator (p. 733). Such an occurrence *defines* a lambda expression, which is *evaluated* at runtime to produce a new kind of value: *an instance of a functional interface*. The *(deferred) execution* of the lambda body only occurs at a later time when the sole abstract method of this functional interface instance is invoked.

In the rest of this section, we take a closer look at the parameter list, the lambda body, the type checking, and evaluation of lambda expressions.

Lambda Parameters

The parameter list of a lambda expression is a comma-separated list of formal parameters that is enclosed in parentheses, `()`, analogous to the parameter list in a method declaration. There is one special case where the parentheses can be omitted, which we shall get to shortly. The lambda body in the code examples in this subsection is intentionally kept simple—an empty code block, `{}`, that does nothing.

The formal parameters of a lambda expression can be declared as one of the following forms for parameter declarations:

- *Declared-type parameters*

  If the types of the parameters are explicitly specified, they are known as *declared-type parameters*. The syntax of declared-type parameters is analogous to the formal parameters in a method declaration.

  Click here to view code image

  ```
  (Integer a, String y) -> {};           // Multiple declared-type para
  ```

- *Inferred-type parameters*

  If the types of the parameters are not explicitly specified, they are known as *inferred-type parameters*. Types of the inferred-type parameters are derived from the context, and if necessary, from the functional type that is the target type of the lambda expression, as explained later in this section.

  Click here to view code image

  ```
  (a, b) -> {};                          // Multiple inferred-type para
  ```

  If the types of the parameters are explicitly specified with the reserved type name `var`, their type is inferred by local variable type inference (§3.13, p. 142). Thus the syntax of such a parameter is consistent with the syntax of a local variable declaration. We will refer to such parameters as `var`*-type inferred parameters*.

  Click here to view code image

  ```
  (var a, var b) -> {};                 // Multiple var-type inferred p
  ```

A lambda expression with inferred parameters is called an *implicitly typed lambda expression*.

All parameters in the parameter list must conform to the same form of parameter declaration—mixing of different forms of parameter declaration is not allowed. Parentheses are mandatory with multiple parameters, whether they are declared-type or inferred-type. For a parameter list with a single inferred-type parameter, the parentheses can be omitted—but not for a declared-type or a `var`-type inferred parameter. Also, only declared-type parameters and `var`-type inferred parameters can have modifiers or annotations, like the `final` modifier or an annotation.

The code examples below illustrate the different forms of formal parameter declarations in a lambda expression.

Click here to view code image

```
() -> {};                         // Empty parameter list

// Single formal parameter:
(String str) -> {};               // Single declared-type parameter
(str)        -> {};               // Single inferred-type parameter
str          -> {};               // Single inferred-type parameter
(var str)    -> {};               // Single var-type inferred parameter

// Multiple formal parameters:
(Integer x, Integer y) -> {};     // Multiple declared-type parameters
(x, y)                 -> {};     // Multiple inferred-type parameters
(var x, var y)         -> {};     // Multiple var-type inferred parameters

// Modifiers and annotations with formal parameters:
(final int i, int j) -> {};       // Modifier with declared-type parameters
(final var i, var j) -> {};       // Modifier with var-type inferred parameter
(@NonNull int i, int j) -> {};    // Annotation with declared-type parameter
(var i, @Nullable var j)-> {};    // Annotation with var-type inferred paramet

// Parentheses are mandatory with multiple formal parameters:
String str           -> {};       // Illegal: Missing parentheses
var str              -> {};       // Illegal: Missing parentheses
Integer x, Integer y -> {};       // Illegal: Missing parentheses
x, y                 -> {};       // Illegal: Missing parentheses
var x, var y         -> {};       // Illegal: Missing parentheses

// All formal parameters must be either declared-type, inferred-type, or
// var-type inferred parameters.
(String str, j)     -> {};      // Cannot mix declared-type and inferred-type
(String str, var j) -> {};      // Cannot mix declared-type and var-type infer
(var str, j)        -> {};      // Cannot mix var-type inferred and inferred-t

// Modifiers and annotations cannot be used with inferred-type parameters.
(final str, j)      -> {};      // No modifiers with inferred-type parameters
(str, @NonNull j)   -> {};      // No annotations with inferred-type parameter
```

Lambda Body

A lambda body is either a *single expression* or a *statement block*. Execution of a lambda body either has a non-`void` return (i.e., its evaluation returns a value), or has a `void` return (i.e., its evaluation does not return a value), or its evaluation throws an exception.

A single-expression lambda body is used for short and succinct lambda expressions. A single-expression lambda body with a `void` return type is commonly used to achieve side effects. The `return` keyword is not allowed in a single-expression lambda body.

In the examples below, the body of the lambda expressions is an *expression* whose execution returns a value—that is, has a non-`void` return.

Click here to view code image

```
() -> 2021                                  // Expression body, non-void retur
() -> null                                  // Expression body, non-void retur
(i, j) -> i + j                             // Expression body, non-void retur
(i, j) -> i <= j ? i : j                    // Expression body, non-void retur
str -> str.length() > 3                     // Expression body, non-void retur
str -> str != null                          // Expression body, non-void retur
       && !str.equals("") && str.length() > 3
       &&  str.equals(new StringBuilder(str).reverse().toString())
```

In the following examples, the lambda body is an *expression statement* that can have a `void` or a non-`void` return. However, if the abstract method of the functional interface does not return a value, the non-`void` return of an expression statement body can be interpreted as a `void` return—that is, the return value is ignored.

Click here to view code image

```
val -> System.out.println(val)    // Method invocation statement, void return
sb -> sb.trimToSize()             // Method invocation statement, void return
sb -> sb.append("!")              // Method invocation statement, non-void ret
() -> new StringBuilder("?")      // Object creation statement, non-void retur
value -> value++                  // Increment statement, non-void return
value -> value *= 2               // Assignment statement, non-void return
```

The following examples are not legal lambda expressions:

Click here to view code image

```
(int i) -> while (i < 10) ++i     // Illegal: not an expression but statement
(x, y) -> return x + y            // Illegal: return not allowed in expression
```

The statement block comprises declarations and statements enclosed in curly brackets (`{}`). The `return` statement is only allowed in a block lambda body, and the rules are the same as those in a method body: A `return` statement with an argument can only be used for a non-`void` return and its use is mandatory, whereas a `return` statement with no argument can only be used for a `void` return, but its use is optional. The `return` statement terminates the execution of the lambda body immediately.

Click here to view code image

```
() -> {}                                    // Block body, void return
() -> { return 2021; }                      // Block body, non-void return
() -> { return 2021 }      // Illegal: statement terminator (;) in block missi
() -> { new StringBuilder("Go nuts."); }           // Block body, void return
() -> { return new StringBuilder("Go nuts!"); }    // Block body, non-void ret
```

```
(int i) -> { while (i < 10) ++i; }                  // Block body, void return
(i, j) -> { if (i <= j) return i; else return j; } // Block body, non-void ret
(done) -> {                    // Multiple statements in block body, void ret
  if (done) {

    System.out.println("You deserve a break!");
    return;
  }
  System.out.println("Stay right here!");
}
```

Type Checking and Execution of Lambda Expressions

A lambda expression can only be defined in a context where a functional interface can be used: for example, in an assignment context, a method call context, or a cast context (p. 733). The compiler determines the *target type* that is required in the context where the lambda expression is defined. This target type is always a functional interface type. In the assignment context below, the target type is `Predicate<Integer>`, as it is the target of the assignment statement. Note that the type parameter `T` of the functional interface is `Integer`.

Click here to view code image

```
Predicate<Integer> p1 = i -> i % 2 == 0;  // (1) Target type: Predicate<Intege
```

The *method type* of a method declaration comprises its type parameters, formal parameter types, return type, and any exceptions the method throws.

The *function type* of a functional interface is the method type of its functional method. The target type `Predicate<Integer>` has the following method, where type parameter `T` is `Integer`:

Click here to view code image

```
public boolean test(Integer t);          // Method type: Integer -> boolean
```

The function type of the target type `Predicate<Integer>` is the method type of the this `test()` method:

```
Integer -> boolean
```

The type of the lambda expression defined in a given context must be compatible with the function type of the target type. If the lambda expression has inferred-type parameters, their type is inferred from the context, and if necessary, from the function type.

From the lambda body at (1), it is possible to infer that the type of `i` is `int` (from the expression `i % 2`), and the lambda body evaluates to a `boolean` value (from the evaluation of the `==` operator). Just from the context, it is thus possible to infer that the type of the lambda expression at (1) is:

```
int -> boolean
```

From the function type of the target type `Predicate<Integer>`, the compiler is able to determine that if the inferred `int` type of the parameter `i` in the lambda expression at (1) is promoted to `Integer`, the type of the lambda expression at (1) would then be compatible with the function type of the target type `Predicate<Integer>`, that is:

```
Integer -> boolean
```

The lambda expression defined at (1) above is equivalent to the following implementation using an anonymous class.

Click here to view code image

```
Predicate<Integer> p1 = new Predicate<>() {  // Anonymous class
  public boolean test(Integer i) {
    return i % 2 == 0;
  }
};
```

It is possible to determine that the type of the lambda expression at (1) is compatible with the method type of the only abstract method `test()` defined by the `Predicate<T>` functional interface because it is not ambiguous which method the lambda expression should implement in the functional interface. However, this would not work for an interface that had more than one abstract method, as it would not be clear which abstract method to implement.

In the following assignment, the target type is `java.util.function.IntPredicate`:

Click here to view code image

```
IntPredicate p2 = i -> i % 2 == 0;         // (2) Target type: IntPredicate
```

The `IntPredicate` functional interface has the following abstract method:

Click here to view code image

```
public boolean test(int i);              // Method type: int -> boolean
```

The function type of the target type `IntPredicate` is the method type of its abstract method:

```
int -> boolean
```

The compiler infers that the type of the inferred-type parameter `i` in the lambda expression at (2) should be `int`. As the lambda body returns a `boolean` value, the type of the lambda expression at (2) is

```
int -> boolean
```

The type of the lambda expression is compatible with the function type of the target type `IntPredicate`.

Note that in both examples, the lambda expression is the same, but their types are different in the two contexts. They represent two different values. The type of a lambda expression is determined by the context in which it is defined.

Click here to view code image

```
System.out.println(p1 == p2);           // false
```

The process of type checking a lambda expression in a given context is called *target typing*. The presentation here is simplified, but suffices for our purpose to give an idea of what is involved.

The compiler does the type checking for using lambda expressions. The runtime environment provides the rest of the magic to make it all work. At runtime, the lambda expression is executed when the sole abstract method of the functional interface is invoked.

Click here to view code image

```
boolean result1 = p1.test(2021);          // false
boolean result2 = p2.test(2020);          // true
```

As mentioned earlier, this is an example of deferred execution. Lambda execution is similar to invoking a method on an object. We define a lambda expression as a *function* and use it like a *method*, letting the compiler and the runtime environment put it all together.

Accessing Members in an Enclosing Class

Just like nested blocks, a lambda expression has *lexical or block scope*—that is, names used in a lambda expression are resolved *lexically* in the local context in which the lambda expression is defined.

A lambda expression does *not* inherit names of members declared in the functional interface it implements, which obviously then cannot be accessed in the lambda body.

Since a lambda expression is not a class, there is no notion of the `this` reference. If the `this` reference is used in a lambda expression in a non-static context, it refers to the *enclosing object*, and can be used to access members of this object. The name of a member in the enclosing object has the same meaning when used in a lambda expression. In other words, there are no restrictions to accessing members declared in the enclosing object. In the case of shadowing member names by local variables, the keyword `this` can be explicitly used, and also the keyword `super` to access any members inherited by the enclosing object.

In Example 13.1, the `getPredicate()` method at (7) defines a lambda expression at (8). This lambda expression is defined in a non-static context (i.e., an instance method). This lambda expression accesses the `static` field `strList` and the instance field `banner` in the enclosing class at (1) and (2), respectively.

In the `main()` method in Example 13.1, an `ArrayList<String>` is assigned to the `static` field `strList` at (3) and is initialized. The `ArrayList<String>` referred to by the `static` field `strList` has the following content:

```
[Tom, Dick, Harriet]
```

A `MembersOnly` object is created at (4). Its `StringBuilder` field `banner` is initialized with the string `"love "`. The local variable `obj` declared at (4) refers to this `MembersOnly` object. At (5), a `Predicate<String>` instance is instantiated by calling the `getPredicate()` method on the `MembersOnly` object referred to by the local variable `obj`. This predicate is first executed when the `test()` method is called at (6) on the `Predicate<String>` instance, with the argument string `"never dies!"`. Calling the `test()` method results in the lambda expression created at (5) by the `getPredicate()` method to be executed in the context of the enclosing `MembersOnly` object referred to by the local variable `obj`.

The parameter `str` of the lambda expression is initialized with the string `"never dies!"`—that is, the argument to the `test()` method. In the body of the lambda expression, the `ArrayList<String>` referred to by the `static` field `strList` in the `MembersOnly` class is first printed at (9):

Click here to view code image

```
List: [Tom, Dick, Harriet]
```

At (10), the parameter `str` (with contents `"never dies!"`) is appended to the `StringBuilder` (with contents `"love "`) referred to by the instance field `banner` in the enclosing object, resulting in the following contents in this `StringBuilder`:

```
"love never dies!"
```

Since the length of the string `"never dies!"`, referred to by the parameter `str`, is greater than 5, the lambda expression returns `true` at (11). This is the value returned by the `test()` method call at (6).

In the call to the `println()` method at (6), the argument

Click here to view code image

```
p.test("never dies!") + " " + obj.banner
```

now evaluates as

Click here to view code image

```
true + " " + "love never dies!"
```

Example 13.1 *Accessing Members in the Enclosing Object*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class MembersOnly {

  // Instance variable
  private StringBuilder banner;                                        // (1)

  // Static variable
  private static List<String> strList;                                 // (2)

  // Constructor
  public MembersOnly(String str) {
    banner = new StringBuilder(str);
  }

  // Static method
  public static void main(String[] args) {
    strList = new ArrayList<>();                                       // (3)
    strList.add("Tom"); strList.add("Dick"); strList.add("Harriet");
```

```
      MembersOnly obj = new MembersOnly("love ");                   // (4)
      Predicate<String> p = obj.getPredicate();                     // (5)
      System.out.println(p.test("never dies!") + " " + obj.banner);  // (6)
    }

    // Instance method
    public Predicate<String> getPredicate() {               // (7)
      return str -> {                                       // (8)  Lambda expres
        System.out.println("List: " + MembersOnly.strList);// (9)  Static field
        this.banner.append(str);                            // (10) Instance fiel
        return str.length() > 5;                            // (11) boolean value
      };
    }
  }
```

Output from the program:

Click here to view code image

```
List: [Tom, Dick, Harriet]
true love never dies!
```

Accessing Local Variables in the Enclosing Context

All variable declarations in a lambda expression follow the rules of *block scope*. They are not accessible outside the lambda expression. It also means that we cannot *redeclare* local variables already declared in the enclosing scope. In Example 13.2, redeclaring the parameter `banner` and the local variable `words` at (6) and (7), respectively, in the body of the lambda expression results in a compile-time error. Local variables declared in the enclosing method, including its formal parameters, can be accessed in a lambda expression provided they are *effectively final*. This means that once a local variable has been assigned a value to the time when it is used, its value has not changed during that time. Using the `final` modifier in the declaration of a local variable explicitly instructs the compiler to ensure that this is the case. The `final` modifier implies effectively final. If the `final` modifier is omitted and a local variable is used in a lambda expression, the compiler effectively performs the same analysis as if the `final` modifier had been specified.

A lambda expression may be executed at a later time, after the method in which it is defined has finished execution. At that point, the local variables in its enclosing context that are used in the lambda expression are no longer accessible. To ensure their availability, *copies of their values* are maintained with the lambda expression. This is called *variable capture*, although in essence it is the values that are captured. Note that it is not the object that is copied in the case of a local reference variable, but the reference value. Objects reside on the heap and are accessible via a copy of the reference value. Correct execution of the lambda expression is guaranteed, since these effectively final values cannot change. Note that the state of an object referred to by a `final` or an effectively final reference can change, but not the reference value stored in the reference—that is, such a reference will continue to refer to the same object once it is initialized.

In Example 13.2, the method `getPredicate()` at (1) has one formal parameter (`banner`), and a local variable (`words`) declared at (2). Although the state of the `Array-List<String>` object, referred to by the reference `words`, is changed in the method (we add elements to it), the reference value in the reference does not—that is, it continues to refer to the same object whose reference value it was assigned at (2). The parameter `banner` is assigned the reference value of the argument object when the method is invoked, and continues to refer to this object throughout the method. Both local variables are effectively final. Their values are captured by the lambda expression, and used when the lambda expression is executed at a later time after the call to the `getPredicate()` method in the `main()` method has completed.

However, if we uncomment (3) and (4) in Example 13.2, then both local variables are not effectively final. Their reference values are changed at (3) and (4), respectively. The compiler now flags an error at (8) and (9), respectively because these non-final local variables are used in the lambda expression.

The vigilant reader no doubt will have noticed that no requirement of effectively final was imposed on the field members of the enclosing class or object, when accessed in the lambda body. Reference to the enclosing object or class is captured by the lambda expression, and when the expression is executed at a later time, the reference can readily be used to access values of any fields referenced in the lambda body—no copies of such values need be made, thereby making the effectively final rule unnecessary for accessing fields in the enclosing context.

Example 13.2 *Accessing Local Variables in the Enclosing Method*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LocalsOnly {

  public static void main(String[] args) {
    StringBuilder banner = new StringBuilder("love ");
    LocalsOnly instance = new LocalsOnly();
    Predicate<String> p = instance.getPredicate(banner);
    System.out.println(p.test("never dies!") + " " + banner);
  }

  public Predicate<String> getPredicate(StringBuilder banner) {   // (1)
    List<String> words = new ArrayList<>();                       // (2)
    words.add("Tom"); words.add("Dick"); words.add("Harriet");

//  banner = new StringBuilder();         // (3) Illegal: Not effectively fina
//  words = new ArrayList<>();            // (4) Illegal: Not effectively fina

    return str -> {                                 // (5) Lambda expression
//    String banner = "Don't redeclare me!";        // (6) Illegal: Redeclared
//    String[] words = new String[6];               // (7) Illegal: Redeclared
      System.out.println("List: " + words);         // (8) Local variable
      banner.append(str);                           // (9) Parameter
      return str.length() > 5;
    };
  }
}
```

Output from the program:

Click here to view code image

```
List: [Tom, Dick, Harriet]
true love never dies!
```

13.3 Lambda Expressions and Anonymous Classes

As we have seen in this chapter so far, both anonymous classes and lambda expressions can be used to provide implementation of functional interfaces. Example 13.3 illustrates using both anonymous classes and lambda expressions for this purpose.

A common operation on elements in a collection is to select those elements that satisfy certain criteria. This operation is called *filtering*. Given a list of words, we would like to filter this list for one-word *palindromes*—that is, words spelled the same forward and backward. For example, `"anana"` is a palindrome, but `"banana"` is not.

Example 13.3 defines the generic method `filterList()` at (7) for filtering a list. Its first parameter is a list of type `T` to filter, and the filtering criteria is given by the second parameter which is a generic functional interface. This functional interface, `Predicate<T>`, specifies the `boolean` abstract method `test(T t)`. The argument passed to the method must implement the `Predicate<T>` interface, supplying the implementation of the `boolean` method `test()` that actually determines if an element satisfies the criteria. The `test()` method is an example of a *predicate*—that is, a function that takes an argument and returns a `boolean` value.

The following `boolean` expressions are used to determine whether a word (given by the reference `str`) is a case-sensitive or case-insensitive palindrome, respectively:

Click here to view code image

```
str.equals(new StringBuilder(str).reverse().toString())
str.equalsIgnoreCase(new StringBuilder(str).reverse().toString())
```

Whether a string is a palindrome is determined by comparing the string for equality with the result of reversing the string using a `StringBuilder`.

Example 13.3 *Filtering an* `ArrayList<E>`

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FunWithPalindromes {

  public static void main(String[] args) {

    List<String> words = new ArrayList<>();
    words.add("Otto"); words.add("ADA"); words.add("Alyla");
    words.add("Bob"); words.add("HannaH"); words.add("Java");
    System.out.println("List of words:                " + words);

    System.out.println("-----------Using Anonymous Classes--------------------
    // Use an anonymous class to filter for palindromes (case sensitive).  (1)
    List<String> palindromesA = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equals(new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-sensitive palindromes:   " + palindromesA);

    // Use an anonymous class to filter for palindromes (case insensitive). (2
    List<String> palindromesB = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equalsIgnoreCase(
                           new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-insensitive palindromes: " + palindromesB);
    System.out.println("-----------Using Lambda Expressions-------------------
    Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (
    List<String> palindromes1 = filterList(words, predicate1);            // (
    System.out.println("Case-sensitive palindromes:   " + palindromes1);

    Predicate<String> predicate2 = str ->
        str.equalsIgnoreCase(new StringBuilder(str).reverse().toString());// (
    List<String> palindromes2 = filterList(words, predicate2);            // (
    System.out.println("Case-insensitive palindromes: " + palindromes2);
  }

  /**
   * Filters a list according to the criteria of the predicate.
   * @param list       List to filter
   * @param predicate  Provides the criteria for filtering the list
   * @return           List of elements that match the criteria
   */
  public static <E> List<E> filterList(List<E> list,                      // (
                                       Predicate<E> predicate) {
    List<E> result = new ArrayList<>();
    for (E element : list) {
      if (predicate.test(element)) {                                      // (
        result.add(element);
      }
    }
    return result;
  }
}
```

Output from the program:

Click here to view code image

```
List of words:                [Otto, ADA, Alyla, Bob, HannaH, Java]
-----------Using Anonymous Classes---------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
-----------Using Lambda Expressions--------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
```

Filtering Criteria Defined by Anonymous Classes

Example 13.3 uses anonymous classes to instantiate the criteria object, as shown at (1) and (2). The basic idea is that we can both declare and instantiate the class at the same time, where it is needed in the code, and in our case, as an argument in the call to the `filterList()` method. The type parameter `E` in this case is `String`. The anonymous classes at (1) and (2) provide implementations of the `test()` method for strings. The method is called at (8) to determine whether a `String` fulfills the selection criteria.

By using anonymous classes, we have avoided creating separate concrete classes, but the verbosity of declaring anonymous classes to encapsulate a single method is inescapable. And we still have to declare a new anonymous class for each selection criterion, duplicating a lot of boilerplate code.

Filtering Criteria Defined by Lambda Expressions

Ideally we would like to pass the code for the selection criteria as an argument to the `filterList()` method so that the method can apply the criteria to the elements in the list—that is, be able to change the behavior of the `filterList()` method depending on the selection criteria. This is an example of *behavior parameterization*.

Knowing that something is a `Predicate<T>`, all the information about the abstract method it implements can be inferred, as it is the only abstract method in the interface: its name, its parameters, any value it returns, and whether it throws any exceptions.

The assignment at (3) in Example 13.3 uses a lambda expression to provide an implementation for the parameterized `Predicate<String>` functional interface:

Click here to view code image

```
Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (
```

The reference `predicate1` on the left-hand side is of type `Predicate<String>`, and it is assigned the value of the lambda expression on the right-hand side.

The lambda expression at (3) defines an anonymous function that takes a `String` as the only parameter, and returns a `boolean` value. Its type is `String -> boolean`. Recall that the `test()` method of the `Predicate<String>` functional interface type does exactly that. The function type of the `Predicate<String>` functional interface is also `String -> boolean`. The compiler can type check that the lambda expression is assignable to the reference on the left-hand side, since the expression represents an anonymous function that is compatible with the sole abstract method `test()` of the parameterized `Predicate<String>` interface.

The lambda expression at (3) is passed as an argument to the `filterList()` method via the reference `predicate1` at (4). It is only executed when the `test()` method is called with a `String` argument in the `filterList()` method at (8).

The anonymous class declaration at (1):

Click here to view code image

```
new Predicate<String>() {
  @Override public boolean test(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
  }
}
```

is implemented by the lambda expression at (3):

Click here to view code image

```
str -> str.equals(new StringBuilder(str).reverse().toString())
```

Now we need only pass a new lambda expression to the `filterList()` method to filter a list based on selection criteria. Using lambda expressions is more precise, concise, and readable than using anonymous classes.

Later, when we discuss *streams* (Chapter 16, p. 879), we will also do away with the `filterList()` method for filtering lists.

Lambda Expressions versus Anonymous Classes

Implementation

A lambda expression can only be used to provide implementation of exactly one functional interface. It represents an anonymous function. Unlike an object, it has only behavior and no state.

An anonymous class is restricted to either implementing one interface or extending one class, but it is not restricted to implementing only one abstract method from its supertype.

No separate class file with Java bytecode is created for a lambda expression, in contrast to a separate class file for each anonymous class declared in a source file.

Scope

Lambda expressions do not introduce a new naming scope, and follow the lexical scope rules for nested blocks. Names in a lambda expression are resolved lexically in its enclosing block and enclosing class.

An anonymous class introduces a new naming scope, where names are resolved according to the inheritance hierarchy of the anonymous class, its local enclosing block, and its enclosing class.

Accessing Inherited Members from the Inheritance Hierarchy

Members in the functional interface implemented by a lambda expression are not accessible in the lambda body.

Since an anonymous class can declare and inherit members from its supertype, instances of anonymous classes can have *state*. The `this` and `super` references can be used to access members in the current instance of the anonymous class and its superclass object, respectively.

Accessing Local Declarations in the Enclosing Block

An anonymous class and a lambda expression can only access effectively final local variables in their enclosing local context.

A local variable in a lambda expression cannot shadow a local variable with the same name in the local context because local variables cannot be redeclared. A field name in the anonymous class can shadow a local variable with the same name in the local context.

Accessing Members in the Enclosing Class

A local variable in a lambda expression and a member (either inherited or declared) in an anonymous class can hide a member with the same name in the enclosing class.

A lambda expression and an anonymous class can access any non-hidden members in the enclosing class by their simple names.

In a non-static context, a lambda expression and an anonymous class can access any hidden members in the enclosing class by the `this` reference and the *qualified* `this` reference, respectively.

In a static context, a lambda expression and an anonymous class can access any hidden members in the enclosing class by their qualified names.

Best Practices

Defining an anonymous class can be verbose. Even an implementation of a single method requires a lot of boilerplate code to encapsulate the method in a class definition, with the added risk of making the code hard to read and understand.

The obvious choice for implementing functional interfaces is lambda expressions. Anything beyond that, and there is little choice but to bring in the anonymous classes.

Review Questions

13.1 Which statement is true about functional interfaces and lambda expressions?

Select the one correct answer.

a. A functional interface can only be implemented by lambda expressions.
b. A functional interface declaration can have only one method declaration.
c. In the body of a lambda expression, only `public` members in the enclosing class can be accessed.
d. In the body of a lambda expression, all local variables in the enclosing scope can be accessed.
e. None of the above

13.2 Which of the following statements are true about the following code?

Click here to view code image

```
interface Funky1    { void    absMethod1(String s); }
interface Funky2    { String  absMethod2(String s); }

public class RQ12A99 {
  public static void main(String[] args) {

    Funky1 p1;
    p1 = s -> System.out.println(s);            // (1)
    p1 = s -> s.length();                       // (2)
    p1 = s -> s.toUpperCase();                  // (3)
    p1 = s -> { s.toUpperCase(); };             // (4)
    p1 = s -> { return s.toUpperCase(); };      // (5)

    Funky2 p2;
    p2 = s -> System.out.println(s);            // (6)
    p2 = s -> s.length();                       // (7)
    p2 = s -> s.toUpperCase();                  // (8)
    p2 = s -> { s.toUpperCase(); };             // (9)
    p2 = s -> { return s.toUpperCase(); };      // (10)
  }
}
```

Select the four correct answers.

a. (1) will fail to compile.
b. (2) will fail to compile.
c. (3) will fail to compile.
d. (4) will fail to compile.
e. (5) will fail to compile.
f. (6) will fail to compile.
g. (7) will fail to compile.
h. (8) will fail to compile.
i. (9) will fail to compile.
j. (10) will fail to compile.

13.3 Which statement is true about the following code?

Click here to view code image

```
interface AgreementA { void doIt(); }
interface AgreementB extends AgreementA {}
interface AgreementC extends AgreementB {
  void doIt();
  boolean equals(Object obj);
}

class Beta implements AgreementB {
  public void doIt() {
    System.out.print("Jazz|");
  }
}

public class RQ12A999 {
  public static void main(String[] args) {
    AgreementA a = () -> System.out.print("Java|");         // (1)
    AgreementB b = () -> System.out.print("Jive|");         // (2)
    AgreementC c = () -> System.out.print("Jingle|");       // (3)
    Object o = a = c;                                       // (4)
    b = new Beta();                                         // (5)
    a.doIt();                                               // (6)
    b.doIt();                                               // (7)
    c.doIt();                                               // (8)
    ((AgreementA) o).doIt();                                // (9)
  }
}
```

Select the one correct answer.

a. The program will fail to compile.
b. The program will throw a `ClassCastException`.
c. The program will print `Jingle|Jingle|Jazz|Jingle|`.
d. The program will print `Jingle|Jazz|Jingle|Jingle|`.
e. The program will print `Jingle|Jingle|Jingle|Jazz|`.

13.4 Overview of Built-In Functional Interfaces

Earlier in this chapter, specialized interfaces (including some functional ones) were mentioned that are readily available in the Java SE Platform API (p. 678). To facilitate defining common functions with lambda expressions, the Java SE Platform API also provides a versatile set of functional interfaces for this purpose.

The main support for functional interfaces is found in the `java.util.function` package. The general-purpose generic functional interfaces shown in Table 13.1 represent the four basic operations that are commonly implemented by functions: to *get* a value (`Supplier<T>`), to *test* a predicate (`Predicate<T>`), to *accept* a value but not return a result (`Consumer<T>`), and to *apply* a function to a value in order to compute a new result (`Function<T, R>`).

The term *arity* refers to the number of arguments that a method requires as its input. A method is called *zero-arity*, *one-arity*, or *two-arity*, depending on whether the method has *zero*, *one*, or *two* arguments, respectively. Depending on whether the functional method is zero-arity, one-arity, or two-arity, its functional interface is likewise referred to as zero-arity, one-arity, or two-arity, respectively. Note that the arity of the functional interface reflects the arity of its functional method. Particularly for a generic functional interface, the arity should *not* be confused with the number of type parameters specified for the generic functional interface.

In Table 13.1, except for the `Supplier<T>` functional interface which has a zero-arity functional method, the functional methods for the other three basic functional interfaces are one-arity methods. Accordingly, the `Supplier<T>` functional interface is a zero-arity functional interface, whereas the other functional interfaces are onearity functional interfaces.

Table 13.1 *Basic Functional Interfaces in the* `java.util.function` *Package*

Functional interface	Function type	Arity	
`Supplier<T>`	`() -> T`	Zero-arity	Provide an instance of a `T`.
`Predicate<T>`	`T -> boolean`	One-arity	Evaluate a predicate on a `T`.
`Consumer<T>`	`T -> void`	One-arity	Perform action on a `T`.
`Function<T, R>`	`T -> R`	One-arity	Transform a `T` to an `R`.

It is important to understand the abstract operations that the basic functional interfaces provide before tackling the specialized versions of these functional interfaces in the `java.util.function` package. Since the package provides a wide range of functional interfaces for various purposes, defining new ones should hardly be necessary.

The complete list of all built-in functional interfaces in the `java.util.function` package is given in Table 13.2. The table also shows any `default` methods that a built-in functional interface defines. The idea is not to memorize them all, but to understand how they are categorized according to the four basic functional interfaces in Table 13.1. The specialized versions of the basic functional interfaces are derived by combining one or more of the following three forms:

- *Two-arity specializations of the basic functional interfaces*

  These functional interfaces (`BiPredicate<T,U>`, `BiConsumer<T,U>`, `BiFunction<T,U,R>`) are two-arity specialized counterparts to the corresponding basic functional interface, except for the `Supplier<T>` interface which does not have a two-arity specialization.

- *Extended versions of the* `Function<T,R>` *and* `BiFunction<T,U,R>` *interfaces*

  The functional interfaces `UnaryOperator<T>` and `BinaryOperator<T>` *extend* the `Function<T,T>` and `BiFunction<T,T,T>` interfaces, respectively. As their names imply, the two specialized functional interfaces `UnaryOperator<T>` and `BinaryOperator<T>` are one-arity and two-arity functional interfaces as their superinterfaces, respectively, *where the parameters and the result in each have the same type*.

- *Primitive type specializations of generic functional interfaces*

  The primitive type specializations avoid excessive boxing and unboxing of primitive values when such values are used as objects.

  The primitive type counterparts are specializations of each generic functional interface where one or more type parameters are replaced by a primitive type. Primitive type specializations primarily involve one or more of the primitive types `int`, `long`, or `double`.

  The naming scheme uses one or more prefixes in front of the name of a primitive type functional interface to indicate its function type—that is, the type of the parameters and that of the result. For example, `IntPredicate` has the function type `int -> boolean`, whereas `IntToDoubleFunction` has the function type `int -> double`, and `LongBinaryOperator` has the function type `(long, long) -> long`.

Table 13.2 *Built-In Functional Interfaces in the* `java.util.function` *Package*

Functional interface	**Functional method**	**Function type**	**Default methods unless otherwise indicated**
`Supplier<T>`	`get()`	`() -> T`	
`IntSupplier`	`getAsInt()`	`() -> int`	
`LongSupplier`	`getAsLong()`	`() -> long`	
`DoubleSupplier`	`getAsDouble()`	`() -> double`	
`BooleanSupplier`	`getAsBoolean()`	`() -> boolean`	
`Predicate<T>`	`test(T)`	`T -> boolean`	`and()`, `or()`, `negate()`, `static isEqual()`, `static not()`
`IntPredicate`	`test(int)`	`int -> boolean`	`and()`, `or()`, `negate()`
`LongPredicate`	`test(long)`	`long -> boolean`	`and()`, `or()`, `negate()`
`DoublePredicate`	`test(double)`	`double -> boolean`	`and()`, `or()`, `negate()`
`BiPredicate<T, U>`	`test(T, U)`	`(T, U) -> boolean`	`and()`, `or()`, `negate()`
`Consumer<T>`	`accept(T)`	`T -> void`	`andThen()`
`IntConsumer`	`accept(int)`	`int -> void`	`andThen()`
`LongConsumer`	`accept(long)`	`long -> void`	`andThen()`
`DoubleConsumer`	`accept(double)`	`double -> void`	`andThen()`
`BiConsumer<T, U>`	`accept(T, U)`	`(T, U) -> void`	`andThen()`
`ObjIntConsumer<T>`	`accept(T, int)`	`(T, int) -> void`	
`ObjLongConsumer<T>`	`accept(T, long)`	`(T, long) -> void`	
`ObjDoubleConsumer<T>`	`accept(T, double)`	`(T, double) -> void`	
`Function<T, R>`	`apply(T)`	`T -> R`	`compose()`, `andThen()`, `static identity()`
`IntFunction<R>`	`apply(int)`	`int -> R`	
`LongFunction<R>`	`apply(long)`	`long -> R`	
`DoubleFunction<R>`	`apply(double)`	`double -> R`	
`ToIntFunction<T>`	`applyAsInt(T)`	`T -> int`	
`ToLongFunction<T>`	`applyAsLong(T)`	`T -> long`	
`ToDoubleFunction<T>`	`applyAsDouble(T)`	`T -> double`	
`IntToLongFunction`	`applyAsLong(int)`	`int -> long`	
`IntToDoubleFunction`	`applyAsDouble(int)`	`int -> double`	
`LongToIntFunction`	`applyAsInt(long)`	`long -> int`	
`LongToDoubleFunction`	`applyAsDouble(long)`	`long -> double`	
`DoubleToIntFunction`	`applyAsInt(double)`	`double -> int`	
`DoubleToLongFunction`	`applyAsLong(double)`	`double -> long`	
`BiFunction<T, U, R>`	`apply(T, U)`	`(T, U) -> R`	`andThen()`
`ToIntBiFunction<T, U>`	`applyAsInt(T, U)`	`(T, U) -> int`	
`ToLongBiFunction<T, U>`	`applyAsLong(T, U)`	`(T, U) -> long`	
`ToDoubleBiFunction<T, U>`	`applyAsDouble(T, U)`	`(T, U) -> double`	
`UnaryOperator<T> extends Function<T,T>`	`apply(T)`	`T -> T`	`compose()`, `andThen()`, `static identity()`
`IntUnaryOperator`	`applyAsInt(int)`	`int -> int`	`compose()`, `andThen()`, `static identity()`
`LongUnaryOperator`	`applyAsLong(long)`	`long -> long`	`compose()`, `andThen()`, `static identity()`
`DoubleUnaryOperator`	`applyAsDouble(double)`	`double -> double`	`compose()`, `andThen()`, `static identity()`
`BinaryOperator<T> extends BiFunction<T,T,T>`	`apply(T, T)`	`(T, T) -> T`	`andThen()`, `static maxBy()`, `static minBy()`
`IntBinaryOperator`	`applyAsInt(int, int)`	`(int, int) -> int`	
`LongBinaryOperator`	`applyAsLong(long, long)`	`(long, long) -> long`	
`DoubleBinaryOperator`	`applyAsDouble(double, double)`	`(double, double) -> double`	

The columns in Table 13.3 list the built-in functional interfaces in the `java.util.function` package according to each category of basic functional interface.

Table 13.3 *Summary of Built-In Functional Interfaces*

Supplier<T> **Function<T,R>** **UnaryOperator<T>**			
IntSupplier IntFunction<R> IntUnaryOperator			
LongSupplier LongFunction<R> LongUnaryOperator			
DoubleSupplier DoubleFunction<R> DoubleUnaryOperator			
BooleanSupplier ToIntFunction<T>			
ToLongFunction<T>			
ToDouble-Function<T>			
IntToLong-Function			
IntToDouble-Function			
LongToInt-Function			
LongToDouble-Function			
DoubleToInt-Function			
DoubleToLong-Function			
BiFunction<T,U,R> **BinaryOperator<T>**			
ObjIntConsumer<T> ToIntBiFunction<T,U>			
ObjLongConsumer<T> ToLongBiFunction<T,U>			
ObjDoubleConsumer<T> ToDoubleBiFunction<T,U>			

13.5 Suppliers

As the name suggests, the `Supplier<T>` functional interface represents a supplier of values. From Table 13.4, we see that its functional method `get()` has the type `() -> T`—that is, it takes no argument and returns a value of type `T`.

Table 13.4 shows all supplier functional interfaces provided in the `java.util.function` package. Apart from the functional method shown in Table 13.4, these functional interfaces do not define any additional methods.

Table 13.4 *Suppliers*

Functional interface **Functional method**	
`Supplier<T>` `get:`	`() -> T`
`IntSupplier` `getAsInt:`	`() -> int`
`LongSupplier` `getAsLong:`	`() -> long`
`DoubleSupplier` `getAsDouble:`	`() -> double`
`BooleanSupplier` `getAsBoolean:`	`() -> boolean`

A supplier typically generates, creates, or produces values. Example 13.4 illustrates defining and using suppliers.

The supplier at (1) in Example 13.4 will always create a `StringBuilder` from the string `"Howdy"`. The `StringBuilder` is not created until the `get()` method of the supplier is called.

Click here to view code image

```
Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   // (1)
System.out.println(createSB.get());                            // Prints: Howdy!

String str = "uppercase me!";
Supplier<String> makeUppercase = () -> str.toUpperCase();              // (2)
System.out.println(makeUppercase.get());              // Prints: UPPERCASE ME!
```

The supplier at (2) returns a string that is an uppercase version of the string on which the method `toUppercase()` is invoked. Note that the value of the reference `str` is captured at (2) when the lambda expression is defined and the reference `str` is effectively final. Calling the `get()` method of the supplier results in the `toUppercase()` method being invoked on the `String` instance referenced by the reference `str`.

In the examples below, we use a pseudorandom number generator to define a supplier that can return integers between different ranges. The `intSupplier` below generates a number between 0 (inclusive) and 100 (exclusive).

Click here to view code image

```
Random numGen = new Random();

Supplier<Integer> intSupplier = () -> numGen.nextInt(100); // numGen effect. f
System.out.println(intSupplier.get());             // Prints a number in [0, 1
```

The generic method `listBuilder()` at (12) can be used to build a list of specified length, where the specified `supplier` generates a value every time the `get()` method is called at (13).

The code below builds a list of `Integer` with five values between 0 (inclusive) and 100 (exclusive) by calling the `listBuilder()` method.

Click here to view code image

```
List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
```

Primitive Type Specializations of `Supplier<T>`

The primitive type versions of the generic supplier interface are appropriately named with a prefix to indicate the type of primitive value returned by their functional methods. For example, the integer supplier is named `IntSupplier`. Their functional methods are also appropriately named with a postfix to indicate the type of value they return. For example, the functional method of the `IntSupplier` interface is named `getAsInt`. These primitive type versions are *not* subinterfaces of the generic `Supplier<T>` interface. `BooleanSupplier` is the only specialization with a primitive type other than `int`, `long`, or `double` in the `java.util.function` package.

Example 13.4 also illustrates defining and using suppliers that return primitive values. Non-generic `int` suppliers are used in the following examples, without the overhead of boxing and unboxing `int` values. Calling the `getAsInt()` method results in the lambda expression to be executed.

Click here to view code image

```
IntSupplier intSupplier2 = () -> numGen.nextInt(100);
System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 100).
```

The method `roleDice()` at (14) prints statistics of rolling a many-sided dice a specified number of times using an `IntSupplier` as a dice roller. In the call below, a six-sided dice is rolled 100,000 times using the specified `int` supplier that generates numbers from 1 to 6.

Click here to view code image

```
roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
```

The reader is encouraged to work through Example 13.4, as it provides additional examples of defining and using suppliers.

Example 13.4 *Implementing Suppliers*

Click here to view code image

```
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.function.DoubleSupplier;
import java.util.function.IntSupplier;
import java.util.function.Supplier;

public class SupplierClient {
  public static void main(String[] args) {

    Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   //
    System.out.println(createSB.get());                                   // Prints: Ho

    String str = "uppercase me!";
    Supplier<String> makeUppercase = () -> str.toUpperCase();                //
    System.out.println(makeUppercase.get());              // Prints: UPPERCASE

    // Pseudorandom number generator captured and used in lambda expressions:
    Random numGen = new Random();

    // Generate a number between 0 (inclusive) and 100 (exclusive):
    Supplier<Integer> intSupplier = () -> numGen.nextInt(100);
    System.out.println(intSupplier.get());         // Prints a number in [0, 1

    // Build a list of Integers with values between 0 (incl.) and 100 (excl.):
    List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
    System.out.println(intRefList);

    // Build a list of StringBuilders:
    List<StringBuilder> stringbuilderList = listBuilder(6,
        () -> new StringBuilder("str" + numGen.nextInt(10)));           // [0,
    System.out.println(stringbuilderList);

    // Build a list that has the same string:
    List<String> stringList2 = listBuilder(4, () -> "Mini me");
    System.out.println(stringList2);

    // Build a list of LocalTime:
    List<LocalTime> dateList1 = listBuilder(3, () -> LocalTime.now());
    System.out.println(dateList1);

    // Generate a number between 0 (inclusive) and 100 (exclusive):
    IntSupplier intSupplier2 = () -> numGen.nextInt(100);
    System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 1

    // Role many-sided dice:
    roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
    roleDice(8, 1_000_000, () -> 1 + (int) (Math.random() * 8));

    // Build an array of doubles with values
    // between 0.0 (incl.) and 5.0 (excl.):
    DoubleSupplier ds = () -> Math.random() * 5;                     // [0.0,
    double[] dArray = new double[4];
    for (int i = 0; i < dArray.length; i++) {
      dArray[i] = ds.getAsDouble();
    }
    System.out.println(Arrays.toString(dArray));
  }

  /**
   * Creates a list whose elements are supplied by a Supplier<T>.
   * @param num       Number of elements to put in the list.
   * @param supplier  Supplier that supplies a value to put in the list
   * @return          List created by the method
   */
  public static <T> List<T> listBuilder(int num, Supplier<T> supplier) {   //
    List<T> list = new ArrayList<>();
    for (int i = 0; i < num; ++i) {
      list.add(supplier.get());                                            //
    }
    return list;
  }

  /**
   * Print statistics of rolling a many-sided dice the specified
   * number of times using an IntSupplier as dice roller.
   */
```

```
    public static void roleDice(int numOfSides, int numOfTimes,
                                IntSupplier diceRoller) {
      int[] frequency = new int[numOfSides + 1];         // frequency[0] is igno
      for (int i = 0; i < numOfTimes; i++) {
        ++frequency[diceRoller.getAsInt()];                              //
      }
      System.out.println(Arrays.toString(frequency));
    }
  }
```

Probable output from the program:

Click here to view code image

```
Howdy!
UPPERCASE ME!
83
[15, 24, 48, 3, 16]
[str8, str0, str6, str8, str6, str7]
[Mini me, Mini me, Mini me, Mini me]
[15:32:05.707, 15:32:05.707, 15:32:05.707]
54
[0, 16747, 16723, 16701, 16607, 16637, 16585]
[0, 124918, 124385, 125038, 125451, 124618, 124600, 125230, 125760]
[0.2129971975975531, 0.6933477140020566, 1.3559818256541756, 1.183773498854187
```

13.6 Predicates

The `Predicate<T>` interface should be familiar by now, having been used earlier in Example 13.1, Example 13.2, and Example 13.3.

The `Predicate<T>` interface defines a `boolean`-valued function in terms of an instance of its type parameter `T`. From Table 13.5, we see that its functional method `test()` has the type `T -> boolean`—that is, it takes an argument of type `T` and returns a value of type `boolean`.

Table 13.5 shows all the predicate functional interfaces provided in the `java.util.function` package. In addition to the three primitive type predicates recognized by their characteristic prefixes, there is also a generic two-arity specialization (`BiPredicate<T,U>`). Apart from the functional method `test()` shown for each predicate in Table 13.5, these functional interfaces also define `default` methods. Neither the primitive type predicates nor the two-arity predicate are subinterfaces of the `Predicate<T>` interface.

Table 13.5 *Predicates*

Functional interface	**Functional method**	**Default methods unless otherwise indicated**
`Predicate<T>`	`test: T -> boolean`	`and(), or(), negate(), static isEqual(), static not()`
`IntPredicate`	`test: int -> boolean`	`and(), or(), negate()`
`LongPredicate`	`test: long -> boolean`	`and(), or(), negate()`
`DoublePredicate`	`test: double -> boolean`	`and(), or(), negate()`
`BiPredicate<T,U>`	`test: (T, U) -> boolean`	`and(), or(), negate()`

The code below illustrates the `removeIf()` method from the `ArrayList<E>` class that requires a predicate and removes all elements from the list that satisfy the predicate. All palindromes are removed from the list denoted by the reference `words`. The method call `element.test(isPalindrome)` is invoked on each element of the list by the `removeIf()` method.

Click here to view code image

```
Predicate<String> isPalindrome
    = str -> new StringBuilder(str).reverse().toString().equalsIgnoreCase(str)
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isPalindrome);          // Remove all palindromes.
// After:  [Alyah, Java]
```

We can equally implement the `Predicate<T>` functional interface passed as an argument to the `ArrayList.removeIf()` method using an anonymous class:

Click here to view code image

```
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(new Predicate<String>() {
  public boolean test(String str) {
    return new StringBuilder(str).reverse().toString().equalsIgnoreCase(str);
  }
});
// After:  [Alyah, Java]
```

Similarly, the code below removes all words with even length from the list.

Click here to view code image

```
Predicate<String> isEvenLen = str -> str.length() % 2 == 0;
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isEvenLen);             // Remove all even length words.
// After:  [ADA, Alyah, Bob]
```

And in this example, all words starting with `"A"` are removed from the list.

Click here to view code image

```
Predicate<String> startsWithA = str -> str.startsWith("A");
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(startsWithA);           // Remove all words that start with "A"
// After:  [Otto, Bob, HannaH, Java]
```

Composing Predicates

The predicate interfaces define `default` methods to compose *compound* predicates—that is, to chain together predicates with logical AND and OR operations.

> Click here to view code image
>
> ```
> default Predicate<T> negate()
> ```
>
> Returns a predicate that represents the logical negation of this predicate.
>
> Click here to view code image
>
> ```
> default Predicate<T> and(Predicate<? super T> other)
> ```
>
> Returns a composed predicate that represents a short-circuiting logical AND of this predicate and the predicate specified by the `other` parameter.
> The `other` predicate is only evaluated if this predicate is `true`. Any exceptions thrown during the evaluation of either predicate are conveyed to the caller.
>
> Click here to view code image
>
> ```
> default Predicate<T> or(Predicate<? super T> other)
> ```
>
> Returns a composed predicate that represents a short-circuiting logical OR of this predicate and the predicate specified by the `other` parameter.
> The `other` predicate is only evaluated if this predicate is `false`. Any exceptions thrown during the evaluation of either predicate are conveyed to the caller.
>
> Click here to view code image
>
> ```
> static <T> Predicate<T> not(Predicate<? super T> target)
> ```
>
> This `static` method returns a predicate that is the negation of the specified predicate.
>
> Click here to view code image
>
> ```
> static <T> Predicate<T> isEqual(Object targetRef)
> ```
>
> This `static` method returns a predicate that tests if the argument in the call to the `test()` method and the `targetRef` object are equal—for example, `Predicate.isEqual("Aha").test("aha")`. In contrast to the `Object.equals()` method, this method returns `true` if both the argument and the `targetRef` object are `null`.

At (1) below, the `isPalindrome` predicate is negated to define a predicate that tests if a string is *not* a palindrome.

The method calls in a compound predicate are executed from *left to right* with *short-circuit* evaluation of the predicates. The compound predicate `x.or(y).and(z)` at (2) is evaluated as `((x.or(y)).and(z))`, where `x`, `y`, and `z` are constituent predicates `isEvenLen`, `startsWithA`, and `isNotPalindrome`, respectively. The `or()` method is executed first, followed by the `and()` method. However, the `startsWithA` predicate is only evaluated if the `isEvenLen` predicate was `false`—that is, the `or()` method tests the `isEvenLen` predicate first, but not the `startsWithA` predicate unless the first one is `false`. Analogously, the `isNotPalindrome` predicate is only evaluated if the predicate on which the `and()` method is invoked is `true`. Note that the *same* argument is passed to all the constituent predicates that comprise a compound predicate. Schematically, the evaluation of the method call `composedPredicate.test("Adda")` at (3) proceeds as follows:

Click here to view code image

```
   ((x.or(y)).and(z))
 = ((true.or(y)).and(z))
 = (true.and(z))
 = (true.and(false))
 = false
// A string that is not a palindrome.
Predicate<String> isNotPalindrome = isPalindrome.negate();       // (1)

// A string with even length or starts with an 'A', and is not a palindrome.
Predicate<String> composedPredicate
    = isEvenLen.or(startsWithA).and(isNotPalindrome);            // (2)
System.out.println("Using composed predicate on \"Adda\": "
        + composedPredicate.test("Adda"));                       // (3) false.

// A string with even length, or it starts with an 'A' and is not a palindrome
Predicate<String> conditionalOperators
    = str -> str.length() % 2 == 0 || str.startsWith("A")        // (4)
          && !(new StringBuilder(str).reverse().toString().equalsIgnoreCase(st
System.out.println("Using conditional operators on \"Adda\": "
        + conditionalOperators.test("Adda"));                    // (5) true.
```

The evaluation should be contrasted with the predicate at (4) that is defined with the negation operator `!` and the short-circuit conditional operators `||` and `&&`. Note that the evaluation order is different for the conditional operators because of the precedence rules: `a || b && !c` is evaluated as `(a || (b && (!c)))`, where `a`, `b`, and `c`are `boolean` expressions `str.length() % 2 == 0`, `str.startsWith("A")`, and `(new String-Builder(str).reverse().toString().equalsIgnoreCase(str))`, respectively. Schematically, the evaluation of the method call `conditionalOperators.test("Adda")` at (5) proceeds as follows:

```
   (a || (b && (!c)))
```

```
= (a || (b && (!true)))
= (a || (b && false))
= (a || (false))
= (true || false)
= true
```

Using equality predicates is illustrated by the following examples:

Click here to view code image

```
Predicate<String> isEqualToTarget = Predicate.isEqual("Ada");
System.out.println(isEqualToTarget.test("Adda"));           // false.
System.out.println(Predicate.isEqual("Ada").test("Ada"));   // true.
System.out.println(Predicate.isEqual("null").test("null")); // true.
```

Primitive Type Specializations of `Predicate<T>`

The functional interfaces `IntPredicate`, `LongPredicate`, and `DoublePredicate` evaluate predicates with `int`, `long`, and `double` arguments, respectively, avoiding the overhead of boxing and unboxing of primitive values (see Table 13.5). The primitive type versions are *not* subinterfaces of the `Predicate<T>` interface.

Click here to view code image

```
Predicate<Integer> isEven = i -> i % 2 == 0;         // Operand unboxed.
System.out.println("2021 is an even number: "
                  + isEven.test(2021));              // Argument boxed. false.

IntPredicate isEvenInt = i -> i % 2 == 0;            // No unboxing.
System.out.println("2021 is an even number: "
                  + isEvenInt.test(2021));           // No boxing. false.
```

Each primitive type version also provides methods for negating a predicate and composing predicates using the methods for short-circuiting logical AND and OR operations.

Click here to view code image

```
IntPredicate isOddInt = isEvenInt.negate();          // Negating a predicate.
System.out.println("2020 is an odd number: "
                  + isOddInt.test(2020));            // false.

IntPredicate isInRange = i -> -100 <= i && i <= 100; // Range: [-100, 100]
System.out.println("21 is in range and odd: "
                 + isInRange.and(isOddInt).test(21));// true.
```

Two-Arity Specialization of `Predicate<T>`: `BiPredicate<T, U>`

The `BiPredicate<T, U>` interface is a two-arity specialization of the `Predicate<T>` interface. From Table 13.5, we see that its functional method `test()` has the type `(T, U) -> boolean`—that is, it takes two arguments of type `T` and `U`, and returns a `boolean` value. There are no primitive type specializations of the `BiPredicate<T, U>` interface. Example 13.5 illustrates defining and using two-arity predicates. The following two-arity predicate tests if an element is a member (or is contained) in a list. The reference `filenames` refers to a list of file names.

Click here to view code image

```
BiPredicate<String, List<String>> isMember
    = (element, list) -> list.contains(element);
System.out.println(isMember.test("X-File4.doc", filenames));  // true.
```

The two-arity predicate below determines if a file name has an extension from a specified set of file extensions.

Click here to view code image

```
BiPredicate<String, Set<String>> selector = (filename, extensions) ->
    extensions.contains(filename.substring(filename.lastIndexOf('.')));
System.out.println(selector.test("Y-File.pdf", extSet));      // true.
```

Determining the file extension is generalized in Example 13.5 to a list of file names using the generic method `filterList()` which takes three parameters: a list of file names, a set of file extensions, and a two-arity predicate to do the selection. In the method `filterList()`, for each element in the list, the following method call is executed: `selector.test(element, extSet)`.

The `BiPredicate<T, U>` interface also defines `default` methods to compose compound two-arity predicates. As expected, the `or()` and the `and()` methods require a two-arity predicate as an argument. A simple example is given in Example 13.5 to check if the product or the sum of two numbers is equal to a given number:

Click here to view code image

```
int number = 21;
BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
System.out.println(isProduct.or(isSum).test(7, 3));       // true.
```

Example 13.5 *Implementing the* `BiPredicate<T, U>` *Functional Interface*

Click here to view code image

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.BiPredicate;

public class BiPredicateClient {

  public static void main(String[] args) {

    // List with filenames:
    List<String> filenames = new ArrayList<>();
    filenames.add("X-File1.pdf"); filenames.add("X-File2.exe");
    filenames.add("X-File3.fm"); filenames.add("X-File4.doc");
    filenames.add("X-File5.jpg"); filenames.add("X-File6.jpg");
    System.out.println("Filenames: " + filenames);

    // BiPredicate for membership in a list.
    BiPredicate<String, List<String>> isMember =
        (element, list) -> list.contains(element);
    System.out.println(isMember.test("X-File4.doc", filenames));  // true.
    // Set with file extensions:
    Set<String> extSet = new HashSet<>();
    extSet.add(".pdf"); extSet.add(".jpg");
    System.out.println("Required extensions: " + extSet);

    // BiPredicate to determine if a filename has an extension from a specifie
    // set of file extensions.
    BiPredicate<String, Set<String>> selector = (filename, extensions) ->
        extensions.contains(filename.substring(filename.lastIndexOf('.')));
    System.out.println(selector.test("Y-File.pdf", extSet));      // true.

    List<String> result = filterList(filenames, extSet, selector);
    System.out.println("Files with required extensions: " + result);

    int number = 21;
    BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
    BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
    System.out.println(isProduct.or(isSum).test(7, 3));
  }

  /**
   * Filters a list according to the criteria of the selector.
   * @param list       List to filter
   * @param extSet     Set of file extensions
   * @param selector   BiPredicate that provides the criteria for filtering
   * @return           List of elements that match the criteria
   */
  public static <E, F> List<E> filterList(List<E> list,
                                          Set<F> extSet,
                                          BiPredicate<E, Set<F>> selector) {
    List<E> result = new ArrayList<>();
    for (E element : list)
      if (selector.test(element, extSet))
        result.add(element);
    return result;
  }
}
```

Output from the program:

Click here to view code image

```
Filenames: [X-File1.pdf, X-File2.exe, X-File3.fm, X-File4.doc, X-File5.jpg, X-
File6.jpg]
```

```
true
Required extensions: [.pdf, .jpg]
true
Files with required extensions: [X-File1.pdf, X-File5.jpg, X-File6.jpg]
true
```

13.7 Consumers

The `Consumer<T>` functional interface represents a consumer of values. From Table 13.6, we see that its functional method `accept()` has the type `T -> void`—that is, it takes an argument of type `T` and returns no value (`void`). Typically, it performs some operation on its argument object.

Table 13.6 shows all the consumer functional interfaces, together with their functional method `accept()` and any `default` methods that are provided by the interface. There are three primitive type one-arity specializations of the `Consumer<T>` functional interface, recognized by their characteristic prefixes. The generic two-arity specialization (`BiConsumer<T,U>`) also has three two-arity primitive type specializations. Only the one-arity consumers and the two-arity generic consumer define the `default` method `andThen()`.

Table 13.6 *Consumers*

Functional interface	**Functional method**	**Default methods**
`Consumer<T>`	`accept: T -> void`	`andThen()`
`IntConsumer`	`accept: int -> void`	`andThen()`
`LongConsumer`	`accept: long -> void`	`andThen()`
`DoubleConsumer`	`accept: double -> void`	`andThen()`
`BiConsumer<T,U>`	`accept: (T, U) -> void`	`andThen()`
`ObjIntConsumer<T>`	`accept: (T, int) -> void`	
`ObjLongConsumer<T>`	`accept: (T, long) -> void`	
`ObjDoubleConsumer<T>`	`accept: (T, double) -> void`	

Generally, a consumer performs an operation on its argument object, but does not return a value. The formatter below prints a `double` value with two decimal places. The type of the lambda expression is `Double -> void`, and the lambda expression is executed when the method `accept()` is invoked.

Click here to view code image

```
Consumer<Double> formatter = d -> System.out.printf("Value: %.2f%n", d);
formatter.accept(3.145);                           // Value: 3.15
```

In the code below, the `resizeSB` consumer resizes a `StringBuilder` to length 4—a more flexible resizer is presented a little later. The `reverseSb` consumer reverses the contents of a `StringBuilder`. The `printSB` consumer prints a `StringBuilder`. In each case, the type of the lambda expression is `StringBuilder -> void`.

Click here to view code image

```
StringBuilder sb1 = new StringBuilder("Banana");
Consumer<StringBuilder> resizeSB = sb -> sb.setLength(4);
resizeSB.accept(sb1);                             // Bana
Consumer<StringBuilder> reverseSB = sb -> sb.reverse();
reverseSB.accept(sb1);                            // anaB

Consumer<StringBuilder> printSB
    = sb -> System.out.println("StringBuilder: " + sb);
printSB.accept(sb1);                              // StringBuilder: anaB
```

The `ArrayList.forEach()` method requires a consumer that is applied to each element of the list—that is, the method call `consumer.accept(element)` is executed on each element in the list. The consumer below prints an element of the list `words` in lowercase.

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> System.out.print(s.toLowerCase() + " "));
// otto ada alya bob hannah java
```

The code below implements the `Consumer<String>` interface using an anonymous class which is passed as an argument to the `ArrayList.forEach()` method:

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(new Consumer<String>() {
  public void accept(String s) {

    System.out.print(s.toLowerCase() + " ");
  }
});
// otto ada alya bob hannah java
```

The following consumer prints each element of the list `words` that has an even length:

Click here to view code image

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> {if (s.length() % 2 == 0) System.out.print(s + " ");});
// Otto Alya HannaH Java
```

Composing Consumers

The method `andThen()` can be used to chain together consumers to compose compound consumers. The three consumers used earlier to resize, reverse, and print a `StringBuilder` can be chained together as seen here:

Click here to view code image

```
resizeSB.andThen(reverseSB)
    .andThen(printSB).accept(new StringBuilder("Banana")); // StringBuilder: a
```

The constituent consumers are executed one after the other from *left to right*. Note that the reference to the `StringBuilder` instance passed to the `accept()` method is also passed to each constituent consumer.

> Click here to view code image
>
> `default Consumer<T> andThen(Consumer<? super T> after)`
>
> Returns a composed `Consumer` that evaluates this operation first followed by the specified `after` operation.
> Any exception thrown during the evaluation of either operation aborts the evaluation of the composed operation and the exception is conveyed to the caller.

Primitive Type Specializations of `Consumer<T>`

From Table 13.6, we see that the non-generic functional interfaces `IntConsumer`, `LongConsumer`, and `DoubleConsumer` define the functional method `accept()` that takes an `int`, a `long`, or a `double` value as an argument, respectively, but does not return a value (`void`). The primitive type versions avoid the overhead of boxing and unboxing of primitive values. They are *not* subinterfaces of the `Consumer<T>` interface, and they all provide the `andThen()` method to compose compound primitive type consumers.

The two `IntConsumer`s below print the square root and the square of their argument, respectively. They are chained by the `andThen()` method that requires an `IntConsumer`.

Click here to view code image

```
IntConsumer sqrt = i -> System.out.printf("%.2f%n", Math.sqrt(i));
IntConsumer sqr = i -> System.out.printf("%d%n", i * i);
sqrt.andThen(sqr).accept(15);     // 3.87
                                  // 225
```

Two-Arity Specialization of `Consumer<T>`:

`BiConsumer<T, U>`

The `BiConsumer<T, U>` interface is a two-arity specialization of the `Consumer<T>` interface. From Table 13.6, we see that its functional method `accept()` has the type `(T, U) -> void`—that is, it takes two arguments of type `T` and type `U`, and does not return a value (`void`). The `BiConsumer<T, U>` interface provides the `andThen()` method to create compound two-arity consumers.

The following code illustrates defining and using two-arity consumers.

Click here to view code image

```
BiConsumer<String, Double> formatPrinter
    = (format, obj) -> System.out.printf(format, obj);
formatPrinter.accept("Math.PI:|%10.3f|%n", Math.PI); // Math.PI:|     3.142|
```

The `java.util.Map.forEach()` method requires a two-arity consumer that is applied to each entry `(key, value)` in the map—that is, the method call `biconsumer.accept(key, value)` is executed for each entry in the map. The two-arity consumer below formats and prints all entries in the map given by the reference `strLenMap`. The key in this map is of type `String` and the value is of type `Integer`— that is, `HashMap<String, Integer>`. The value is the length of the key string.

Click here to view code image

```
// Map entries (default format): {Java=4, Bob=3, Otto=4, HannaH=6, Alya=4, ADA
strLenMap.forEach((key, value) -> System.out.printf("(%s:%d) ", key, value));
// (Java:4) (Bob:3) (Otto:4) (HannaH:6) (Alya:4) (ADA:3)
```

Primitive Type Specializations of `BiConsumer<T, U>`

Table 13.6 shows the generic functional interfaces `ObjIntConsumer<T>`, `ObjLongConsumer<T>`, and `ObjDoubleConsumer<T>` that are specializations of the `BiConsumer<T, U>` interface. The functional method `accept()` of these primitive type specializations takes two arguments: One is an object of type `T` and the other is a primitive value. These functional interfaces are *not* subinterfaces of the `BiConsumer<T, U>` interface, and they do not provide `default` methods to chain consumers.

The code below shows a new version of resizing a `StringBuilder` written earlier, where the required length was hard-coded in the lambda expression definition and could not be changed. Using an

`ObjIntConsumer<StringBuilder>`, the required length can be passed as a parameter in the definition of the lambda expression, as shown below:

Click here to view code image

```
ObjIntConsumer<StringBuilder> resizeSB2 = (sb, len) -> sb.setLength(len);
StringBuilder sb2 = new StringBuilder("bananarama");
resizeSB2.accept(sb2, 6);       // The required length passed as a parameter.
System.out.println("StringBuilder resized: " + sb2);
// StringBuilder resized: banana
```

13.8 Functions

The `Function<T, R>` interface represents a function or an operation that transforms an argument object to a result object, where the object types need not be the same. From Table 13.7, we see that its functional method `apply()` has the type `T -> R`—that is, it takes an argument of type `T` and returns a result of type `R`.

Table 13.7 also shows specialized functions for primitive types, together with their functional methods. They do not define any `default` methods.

The `BiFunction<T, U, R>` interface and its primitive type versions are discussed in §13.9, p. 717. The specialized versions `UnaryOperator<T>` and `BinaryOperator<T>`, which provide functions where the arguments and the result are of the same type, are discussed in §13.10, p. 720, and §13.11, p. 721, respectively.

Table 13.7 *Functions*

Functional interface	Functional method	Default methods unless otherwise
`Function<T, R>`	`apply: T -> R`	`compose(), andThen(), static identity()`
`IntFunction<R>`	`apply: int -> R`	
`LongFunction<R>`	`apply: long -> R`	
`DoubleFunction<R>`	`apply: double -> R`	
`ToIntFunction<T>`	`applyAsInt: T -> int`	
`ToLongFunction<T>`	`applyAsLong: T -> long`	
`ToDoubleFunction<T>`	`applyAsDouble: T -> double`	
`IntToLongFunction`	`applyAsLong: int -> long`	
`IntToDoubleFunction`	`applyAsDouble: int -> double`	
`LongToIntFunction`	`applyAsInt: long -> int`	
`LongToDoubleFunction`	`applyAsDouble: long -> double`	
`DoubleToIntFunction`	`applyAsInt: double -> int`	
`DoubleToLongFunction`	`applyAsLong: double -> long`	

Example 13.6 illustrates defining and using functions. The first lambda expression tests whether an integer is in a given range. It has the type `Integer -> Boolean`, compatible with the function

type of the `Function<Integer, Boolean>` interface. Note that it returns a `Boolean`, as opposed to a lambda expression which implements a `Predicate<T>` that always returns a `boolean` value.

Click here to view code image

```
Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
System.out.println("Boolean expression is: " + boolExpr.apply(99));
// Boolean expression is: true

Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
// 24mi = 38.40km
```

The second lambda expression above converts miles to kilometers. It has the type `Integer -> Double`, compatible with the function type of the `Function<Integer, Double>` interface.

The method `listBuilder()` in Example 13.6 creates a list from an array by applying a `Function<T, R>` to each array element. The `Function<T, R>` is passed as an argument to the method.

Click here to view code image

```
String[] strArray = {"One", "Two", "Three", "Four"};
List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(s));
System.out.println("Build StringBuilder list: " + sbList);
// Build StringBuilder list: [One, Two, Three, Four]
```

The example above creates a list of `StringBuilder` from an array of `String`. The signature of the method call can be inferred to be the following:

Click here to view code image

```
listBuilder(String[], String -> StringBuilder)
```

with the type parameters `T` and `R` in the generic type `Function<T, R>` inferred as `String` and `StringBuilder`, respectively, resulting in the parameterized type `Function<String, StringBuilder>`.

The second example creates a list of `Integer`s from an array of `String`s, where the functional interface parameter in the method call is inferred to be `Function<String, Integer>`.

Click here to view code image

```
List<Integer> intList = listBuilder(strArray, s -> s.length());
System.out.println("Build Integer list: " + intList);
// Build Integer list: [3, 3, 5, 4]
```

Example 13.6 *Implementing Functions*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

public class FunctionClient {
  public static void main(String[] args) {

    // Examples of Function<T,R>:
    Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
    System.out.println("Boolean expression is: " + boolExpr.apply(99));
    // Boolean expression is: true

    Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
    System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
    // 24mi = 38.40km

    // Create a list of StringBuilders from an array of Strings.
    String[] strArray = {"One", "Two", "Three", "Four"};
    List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(
    System.out.println("Build StringBuilder list: " + sbList);
    // Build StringBuilder list: [One, Two, Three, Four]

    // Create a list of Integers from an array of Strings.
    List<Integer> intList = listBuilder(strArray, s -> s.length());
    System.out.println("Build Integer list: " + intList);
    // Build Integer list: [3, 3, 5, 4]

    /* Composing unary functions. */
    Function<String, String> f = s -> s + "-One";    // (1)
    Function<String, String> g = s -> s + "-Two";    // (2)

    // Using compose() and andThen() methods.
    System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
    System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
    System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One
    System.out.println();

    System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
    System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
    System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
    System.out.println();

    // Examples of primitive unary functions.
    IntFunction<String> intToStr = i -> Integer.toString(i);
    System.out.println(intToStr.apply(2021));        // 2021
    ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
    System.out.println(strToInt.applyAsInt("2021")); // 2021

    IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
    System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                       37, celsiusToFahrenheit.applyAsDouble(37));
    // 37 Celsius = 98.6 Fahrenheit
}

  /**
   * Create a list from an array by applying a Function to each array element.
   * @param arrayT     Array to use for elements
   * @param func       Function to apply to each array element
   * @return          List that is created
   */
  public static <T, R> List<R> listBuilder(T[] arrayT, Function<T, R> func) {
    List<R> listR = new ArrayList<>();
    for (T t : arrayT) {
      listR.add(func.apply(t));
    }
    return listR;
  }
}
```

Output from the program:

Click here to view code image

```
Boolean expression is: true
24mi = 38.40km
Build StringBuilder list: [One, Two, Three, Four]
Build Integer list: [3, 3, 5, 4]
Three-Two-One
Three-Two-One
Three-Two-One

Three-One-Two
Three-One-Two
Three-One-Two

2021
2021
37 Celsius = 98.6 Fahrenheit
```

Composing Functions

Both the `default` methods `compose()` and `andThen()` of the `Function<T, R>` interface return an instance of a `Function` that is created from the caller function (i.e., the function on which the method is invoked) and the argument function (i.e., the function that is passed as an argument to the method). The two methods differ in the order in which they apply the caller and the argument functions. Given two functions `f` and `g`, the `compose()` and the `andThen()` methods execute as follows:

Click here to view code image

```
f.compose(g).apply(x)  emulates  f.apply(g.apply(x))  or mathematically  f(g(x)).
f.andThen(g).apply(x)  emulates  g.apply(f.apply(x))  or mathematically  g(f(x)).
f.compose(g).apply(x)  and  g.andThen(f).apply(x)  are equivalent.
```

The `compose()` method executes the argument function `g` first and executes the caller function `f` last. The `andThen()` method does the converse: It executes the caller function `f` first and executes the argument function `g` last. Switching the caller and the argument functions of one method in the other method gives the same result. Since the result of one function is passed as an argument to the other function, the return type of the function executed first must be compatible with the parameter type of the function executed last.

Creating compound functions with the `default` methods `compose()` and `andThen()` is illustrated by the code in Example 13.6. Functions `f` and `g` are defined at (1) and (2), respectively. The output from the program shows that the `compose()` method at (3) executes the function `g` first and the function `f` last—the same as the `andThen()` method at (4) with the functions switched and the explicit application at (5). The `andThen()` method at (6) executes the function `f` first and the function `g` last—the same as the `compose()` method at (7) with the functions switched and the explicit application at (8). The return type of the function executed first is also compatible with the parameter type of the function executed last—which is `String` for the functions `f` and `g`.

Click here to view code image

```
Function<String, String> f = s -> s + "-One";     // (1)
Function<String, String> g = s -> s + "-Two";     // (2)

System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One

System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
```

> Click here to view code image
>
> ```
> default <V> Function<V,R> compose(Function<? super V,? extends T> before)
> ```
>
> This generic method returns a composed function that first applies the `before` function to its input, and then applies this function to the result. (This function refers to the function used to invoke the method.)
> Given that the type of this function is `T -> R` and the type of the argument function `before` is `V -> T`, the `compose()` method creates a compound function of type `V -> R`, as the function `before` is executed first and this function last.
>
> Click here to view code image
>
> ```
> default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
> ```
>
> This generic method returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
> Given that the type of this function is `T -> R` and the type of

> the argument function `after` is `R -> V`, the `andThen()` method creates a compound function of type `T -> V`, as this function is executed first and the function `after` last.
> Any exception thrown during the evaluation of either function aborts the evaluation of the composed function and the exception is conveyed to the caller.

Primitive Type Specializations of `Function<T, R>`

As can be seen in Table 13.7, there are three categories of *primitive type one-arity specializations* of the `Function<T, R>` interface, each distinguished by a naming scheme. Also, these primitive type one-arity specializations do not define any `default` methods.

- *`Prim`*`Function<R>`, where *Prim* is either `Int`, `Long`, or `Double`

  These one-arity generic functions have the functional method `apply`: *primitive* `->` `R`, where *primitive* is an `int`, `long`, or `double`—the function takes an argument of primitive type and returns a result of type `R`.

  Click here to view code image

  ```
  IntFunction<String> intToStr = i -> Integer.toString(i);
  System.out.println(intToStr.apply(2021));        // "2021"
  ```

- `To`*`Prim`*`Function<T>`, where *Prim* is either `Int`, `Long`, or `Double`

  These one-arity generic functions have the functional method `applyAs`*Prim*: `T->` *primitive*, where *primitive* is an `int`, `long`, or `double`—the function takes an argument of type `T` and returns a result of primitive type.

  Click here to view code image

  ```
  ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
  System.out.println(strToInt.applyAsInt("2021")); // 2021
  ```

- *Prim*$_1$`To`*Prim*$_2$`Function`, where *Prim*$_1$ and *Prim*$_2$ are `Int`, `Long`, or `Double`, and *Prim*$_1$ `!=` *Prim*$_2$

  These one-arity non-generic functions have the functional method `applyAs`*Prim*$_2$: *primitive*$_1$ `->` *primitive*$_2$, where *primitive*$_1$ and *primitive*$_2$ are `int`, `long`, or `double`, and *primitive*$_1$ `!=` *primitive*$_2$—the function takes an argument of type *primitive*$_1$ and returns a result of type *primitive*$_2$.

  Click here to view code image

```
IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32
System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                  37, celsiusToFahrenheit.applyAsDouble(37));
// 37 Celsius = 98.6 Fahrenheit
```

13.9 Two-Arity Specialization of `Function<T, R>`: `BiFunction<T, U, R>`

The `BiFunction<T, U, R>` interface is the two-arity specialization of the `Function<T, R>` interface. In Table 13.8, we see that its functional method `apply()` has the type `(T, U) -> R`—that is, it takes two arguments of type `T` and `U`, and returns a result of type `R`.

The following code illustrates defining and using two-arity functions. The twoarity function `areaOfRectangle` calculates the area of a rectangle.

Click here to view code image

```
BiFunction<Double, Double, Double> areaOfRectangle
    = (length, width) -> length * width;            // (Double, Double) -> Dou
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

Table 13.8 *Two-arity Functions*

Functional interface	Functional method	[illegible]	Default methods
`BiFunction<T, U, R>`	`apply()`	`(T, U) -> R`	`andThen()`
`ToIntBiFunction<T, U>`	`applyAsInt()`	`(T, U) -> int`	
`ToLongBiFunction<T, U>`	`applyAsLong()`	`(T, U) -> long`	
`ToDoubleBiFunction<T, U>`	`applyAsDouble()`	`(T, U) -> double`	

The two-arity function `concatKeyVal` below concatenates two strings and returns a new string as the result. The reference `map` refers to a `HashMap<String, String>`. The `replaceAll()` method called on this map takes the two-arity function `concatKeyVal` as a parameter, and it replaces the *value* of each entry `(key, value)` in the map with the result of the two-arity function applied to the entry—that is, the method call `concatKeyVal.apply(key, value)` is executed for each entry in the map. The `apply()` method is implemented by the lambda expression below, resulting in the method call `key.concat(value)` being executed for each entry.

Click here to view code image

```
BiFunction<String, String, String> concatKeyVal = (key, val) -> key.concat(val
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(concatKeyVal);
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
```

It is instructive to implement and compare the two-arity function above using an anonymous class:

Click here to view code image

```
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(new BiFunction<String, String, String>() {
  public String apply(String key, String val) {
    return key.concat(val);
  }
});
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
```

Composing Two-Arity Functions

The interface `BiFunction<T, U, R>` provides the `andThen()` method for creating compound two-arity functions. Given a *two-arity function* `f` and a *one-arity function* `g`, the method evaluates the functions as follows:

> `f.andThen(g).apply(x, y)` emulates `g.apply(f.apply(x, y))` —that is, the two-arity function `f` is executed first and the one-arity function `g` last.

In the example below, the functions chained by the `andThen()` method calls are applied from left to right, first the caller function and then the parameter function.

Click here to view code image

```
BiFunction<String, String, String> concatStr = (s1, s2) -> s1 + s2;
Function<String, String> postfix1 = s -> s + "nana";
```

Click here to view code image

```
Function<String, String> postfix2 = s -> s + "s!";
System.out.println(concatStr.andThen(postfix1).andThen(postfix2)
                   .apply("I am going", " ba"));         // I am going bananas
```

In the code below, the `concatStr` two-arity function is both the caller and the parameter function in the call to the `andThen()` method. However, the code does not compile. The reason is easy to understand: The two-arity parameter function requires *two* arguments, but the two-arity caller function can only return a *single* value. A *one-arity* function as the parameter function avoids this problem, as it can accept the single-value result of the caller function. Chaining instances of `BiFunction<T, U, R>` is not as straightforward as chaining instances of `Function<T, R>`.

Click here to view code image

```
BiFunction<String, String, String> concatTwice
               = concatStr.andThen(concatStr);          // Compile-time error
```

> Click here to view code image
>
> ```
> default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)
> ```
>
> This generic method returns a composed two-arity function that first applies this function to its input, and then applies the specified `after` one-arity function to the result.
> Note the type of the parameter: It is `Function` and not `BiFunction`. Given that the type of this two-arity function is `(T, U) -> R` and the type of the one-arity parameter function `after` is `R -> V`, the `andThen()` method creates a compound two-arity function of type `(T, U) -> V`, as this function is executed first and the one-arity function `after` last.
> Any exception thrown during the evaluation of either function aborts the evaluation of the composed function and the exception is conveyed to the caller.

Primitive Type Specializations of `BiFunction<T, U, R>`

Table 13.8 shows that the `BiFunction<T, U, R>` interface has three *primitive type twoarity generic specializations* to `int`, `long`, and `double`, but they do not define any `default` methods for creating compound functions. The specializations are named `To`*`Prim`*`BiFunction<T, U>`, where *Prim* is either `Int`, `Long`, or `Double`. These two-arity generic functions have the functional method `applyAs`*`Prim`*`: (T, U) ->` *primitive*, where *primitive* is an `int`, `long`, or `double`—the function takes two arguments of type `T` and `U`, and returns a result of a primitive type.

In the example below, the `addIntStrs` two-arity function parses two strings as `int` values and returns the sum of the values.

Click here to view code image

```
ToIntBiFunction<String, String> addIntStrs
    = (s1, s2) -> Integer.parseInt(s1) + Integer.parseInt(s2);
System.out.println("10 + 20 = " + addIntStrs.applyAsInt("10", "20"));
// 10 + 20 = 30
```

13.10 Extending `Function<T,T>`: `UnaryOperator<T>`

Table 13.9 shows that the `UnaryOperator<T>` interface *extends* the `Function<T, T>` interface for the special case where the types of the argument and the result are the same. It inherits the functional method `apply()` from the `Function<T, T>` interface. It also inherits the `default` methods `compose()` and `andThen()` from its superinterface `Function<T, T>`, but note that these methods return a `Function<T,T>`, and not a `UnaryOperator<T>`.

Functions where the argument and the result type are the same can easily be refactored to use the `UnaryOperator<T>` interface.

Click here to view code image

```
UnaryOperator<Double> area = r -> Math.PI * r * r;
System.out.printf("Area of circle, radius %.2f: %.2f%n", 10.0, area.apply(10.0
// Area of circle, radius 10.00: 314.16

UnaryOperator<Double> milesToKms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 24.0, milesToKms.apply(24.0));
// 24.00mi = 38.40km
```

The `List.replaceAll(UnaryOperator<E>)` method can be used to replace each elements in the list with the result of applying the specified unary operator to that element. The method replaces all strings in the list `team` with their uppercase versions.

Click here to view code image

```
List<String> team = Arrays.asList("Tom", "Dick", "Harriet");
UnaryOperator<String> toUpper = str -> str.toUpperCase();
team.replaceAll(toUpper);     // [TOM, DICK, HARRIET]
```

Since the `UnaryOperator<T>` interface is a subinterface of the `Function<T, T>` interface and inherits the `default` methods `compose()` and `andThen()`, creating compound unary operators is no different from creating compound functions.

Click here to view code image

```
UnaryOperator<String> f = s -> s + "-One";
UnaryOperator<String> g = s -> s + "-Two";
System.out.println(f.compose(g).apply("Three")); // Three-Two-One
System.out.println(f.andThen(g).apply("Three")); // Three-One-Two
```

Table 13.9 *Unary Operators*

Functional interface	Functional method	Default methods	Static methods unless otherwise indicated
`UnaryOperator<T> extends Function<T,T>`	`apply()`	`compose()`, `andThen()`	`identity()`
`IntUnaryOperator`	`applyAsInt()`, `int -> int`	`compose()`, `andThen()`	`identity()`
`LongUnaryOperator`	`applyAsLong()`, `long -> long`	`compose()`, `andThen()`	`identity()`
`DoubleUnaryOperator`	`applyAsDouble()`, `double -> double`	`compose()`, `andThen()`	`identity()`

Primitive Type Specializations of `UnaryOperator<T>`

The `UnaryOperator<T>` interface has three *primitive type specializations* to `int`, `long`, and `double`. The specializations are named `Prim`UnaryOperator, where *`Prim`* is either `Int`, `Long`, or `Double` (Table 13.9). These non-generic unary operators have the functional method `applyAsPrim`: *`primitive -> primitive`*, where *`primitive`* is an `int`, `long`, or `double`—the operator takes an argument of a primitive type and returns a result of the *same* primitive type.

Click here to view code image

```
DoubleUnaryOperator celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
System.out.printf("%.1f Celsius = %.1f Fahrenheit%n",
                  25.0, celsiusToFahrenheit.applyAsDouble(25.0));
// 25.0 Celsius = 77.0 Fahrenheit

DoubleUnaryOperator kms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 25.0, kms.applyAsDouble(25.0));
// 25.00mi = 40.00km
```

The primitive type unary operators define the `default` methods `compose()` and `andThen()` for creating compound primitive type unary operators. The semantics of these default methods are the same as what we saw earlier (p. 715).

Click here to view code image

```
IntUnaryOperator incrBy1 = i -> i + 1;
IntUnaryOperator multBy2 = i -> i * 2;
System.out.println(incrBy1.compose(multBy2).applyAsInt(4)); // 9
System.out.println(incrBy1.andThen(multBy2).applyAsInt(4)); // 10
```

13.11 Extending `BiFunction<T,T,T>`: `BinaryOperator<T>`

In Table 13.10, we see that the `BinaryOperator<T>` interface *extends* the `BiFunction<T, T, T>` interface for the special case where the types of the two arguments and the result are the same. It inherits the functional method `apply()` from the `BiFunction<T, T, T>` interface, as well as its `andThen()` method.

Click here to view code image

```
BinaryOperator<Double> areaOfRectangle = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
```

```
// 25.00 x 4.00 = 100.00
```

Creating compound binary operators is no different from creating compound twoarity functions using the `andThen()` method, where the parameter function of the method must be a unary operator or a one-arity function.

Click here to view code image

```
BinaryOperator<String> concatTwo = (s1, s2) -> s1 + s2;
UnaryOperator<String> postfix1 = s -> s + "nana";
UnaryOperator<String> postfix2 = s -> s + "s!";
System.out.println(concatTwo.andThen(postfix1).andThen(postfix2)
                     .apply("I am going", " ba"));      // I am going bananas!
```

The two utility methods `maxBy()` and `minBy()` can be used to compare two elements according to a given comparator:

Click here to view code image

```
String maxStr = BinaryOperator.maxBy(String.CASE_INSENSITIVE_ORDER)
                                   .apply("aha", "Madonna");               // Madonna
String minStr = BinaryOperator.minBy(String.CASE_INSENSITIVE_ORDER)
                                   .apply("aha", "Madonna");               // aha
```

Table 13.10 *Binary Operators*

Functional interface	Functional method	Function type	Default methods unless otherwise indicated
`BinaryOperator<T> extends BiFunction<T,T,T>`	`apply`	`(T, T) -> T`	`andThen()`, `static maxBy()`, `static minBy()`
`IntBinaryOperator`	`applyAsInt`	`(int, int) -> int`	
`LongBinaryOperator`	`applyAsLong`	`(long, long) -> long`	
`DoubleBinaryOperator`	`applyAsDouble`	`(double, double) -> double`	

The `BinaryOperator<T>` interface also provides two utility methods to create binary operators for comparing two elements according to a given comparator:

> Click here to view code image
>
> ```
> static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
> ```
>
> Returns a `BinaryOperator` which returns the greater of two elements according to the specified `comparator`.
>
> Click here to view code image
>
> ```
> static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
> ```
>
> Returns a `BinaryOperator` which returns the lesser of two elements according to the specified `comparator`.

Primitive Type Specializations of

`BinaryOperator<T>`

Table 13.10 shows that the `BinaryOperator<T>` interface has three *primitive type specializations* to `int`, `long`, and `double`. The specializations are named `PrimBinaryOperator`, where *Prim* is either an `Int`, `Long`, or `Double` (Table 13.10). These primitive type binary operators have the functional method `applyAsPrim: (primitive, primitive) -> primitive`, where `primitive` is an `int`, `long`, or `double`—the operator takes two arguments of a primitive type and returns a result of the same primitive type.

Click here to view code image

```
DoubleBinaryOperator areaOfRectangle2 = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle2.applyAsDouble(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

13.12 Currying Functions

The functional interfaces in the `java.util.function` package define functional methods that are either one-arity or two-arity methods. For higher arity functional methods, one recourse is to define functional interfaces whose functional method has the desired arity. The functional interface below defines a three-arity functional method—that is, it takes three arguments.

Click here to view code image

```
@FunctionalInterface
interface TriFunction<T, U, V, R> {
  R compute(T t, U u, V v);           // (T, U ,V) -> R
}
```

The `TriFunction<T, U, V, R>` interface can be used to define a lambda expression to calculate the volume of a cuboid.

Click here to view code image

```
// (Double, Double, Double) -> Double
TriFunction<Double, Double, Double, Double> cubeVol = (x, y, z) -> x * y * z;
System.out.println(cubeVol.compute(10.0,  20.0,  30.0));  // 6000.0
```

Another recourse is to apply the technique of *currying* to transform a multi-argument function into a chain of lower arity functions.

The process of currying is illustrated by implementing the three-arity lambda expression above for calculating the volume of a cuboid. Step 1 below derives *a chain of three one-arity functions* that will together compute the volume of a cuboid. At (1), parentheses are used explicitly to show the nested lambdas that *define* each of the one-arity functions—grouping is from right to left. The nesting of the onearity functions is compatible with the nesting of the types in the parameterized functional interface type. An outer function returns its immediate inner function. Step 2 supplies the `x` argument. This is called *partial application*, as it returns a function where the remaining arguments `y` and `z` are still unknown. Step 3 is also partial application, only the `y` argument is supplied, returning a function where now only the `z` argument is unknown. Only at Step 4, when the `z` argument is supplied at (2), can the final one-arity function be executed. The application of the individual onearity functions can also be chained as shown at (3), without going through the intermediate steps.

Click here to view code image

```
// Step 1:
// Partial application: double -> DoubleFunction<DoubleUnaryOperator>
DoubleFunction<DoubleFunction<DoubleUnaryOperator>> uniFuncA
    = (x -> (y -> (z -> x * y * z)));                // (1)

// Step 2:
// Partial application: double -> DoubleUnaryOperator
DoubleFunction<DoubleUnaryOperator> uniFuncB
    = uniFuncA.apply(10.0);                          // 10.0 * y * z

// Step 3:
// Partial application: double -> double
DoubleUnaryOperator uniOpC = uniFuncB.apply(20.0); // 10.0 * 20.0 * z

// Step 4:
// Application:
double vol1 = uniOpC.applyAsDouble(30.0);       // (2) 10.0 * 20.0 * 30.0 = 600
double vol2 = uniFuncA.apply(10.0).apply(20.0).applyAsDouble(30.0); // (3) 600
```

The sequence in which the arguments are supplied in the currying process is irrelevant, and more than one argument can be supplied in each step in accordance with the target type. Each step is a partial application, except for the last one which executes the final function. The process ensures that the number of unknown arguments decreases in each step.

Here we have provided just a taste of currying, but it is a topic worth exploring further. The technique bears the name of Haskell Curry, a famous mathematician and logician of the twentieth century.

13.13 Method and Constructor References

So far we have seen that a Java program can use primitive values, objects (including arrays), and lambda expressions. In this section, we introduce the fourth kind of value that a Java program can use, called *method references* (and their specializations to constructors and array construction). As we shall see, there is a very tight relationship between method references and lambda expressions.

Quite often the body of a lambda expression is just a call to an existing method. The lambda expression simply supplies the arguments required to call and execute the method. In such cases, method references can provide a more concise notation than a lambda expression, potentially increasing the readability of the code. The following code illustrates the relationship between the two notations.

Click here to view code image

```
// String -> void
Consumer<String> outLE = obj -> System.out.println(obj); // (1a)
Consumer<String> outMR = System.out::println;            // (1b)
outMR.accept("Save trees!");                             // (2)
// Calls System.out.println("Save trees!") that prints: Save trees!
```

The lambda expression at (1a):

Click here to view code image

```
obj -> System.out.println(obj)
```

can be replaced by the *method reference* at (1b):

```
System.out::println
```

The method reference above is composed of the *target reference* (`System.out`) on which the method is invoked and the *name of the method* (`println`), separated by the double-colon (`::`) delimiter:

```
targetReference::methodName
```

Note that the target reference precedes the double-colon (`::`) delimiter, and no arguments are specified after the method name. Table 13.11 summarizes the variations in the definition of a method reference which we will discuss in this section.

Table 13.11 *Method and Constructor References*

Purpose of method	Lambda expression/Method reference sy
Designate a static method	`(args) -> RefType.staticMethod(args)` `RefType::staticMethod`

`expr::instanceMethod`	Designates an instance method of a *bounded* instance. Target reference provided by the expr	`(args) -> expr.instanceMethod(args)`
`RefType::instanceMethod`	Designates an instance method of an *unbounded* instance. Target reference provided when the method is invoked.	`(arg0, rest) -> arg0.instanceMethod(rest)`
`ClassType::new`	Designates a constructor. Reference creation of an instance.	`(args) -> new ClassType(args)`
`ElementType[][]...[]::new`	Designates an array constructor. Reference creation of an array.	`arg -> new ElementType[arg][]...[]`

A method reference must have a compatible target type that is a functional interface—a method reference implements an instance of a functional interface, analogous to a lambda expression. The compiler does similar type checking as for lambda expressions to determine whether the method reference is compatible with a given target type.

At (1b) above, the function type of the target type `Consumer<String>` is `String -> void`. The compiler can infer from the target type that the type of the argument passed to the `println()` method must be `String`. The type of the method reference is defined by the type of the method specified in its definition. In this case, the type of the `println()` method is `String -> void`, as it accepts a `String` parameter and does not return a value. Thus the target type `Consumer<String>` is compatible with the method reference `System.out::println`. The reference `outMR` is assigned the value of the method reference at (1b).

Analogous to single-method lambda expressions, method references when executed result in the execution of the method specified in its definition. The instance method `println()` at (1b) accepts a single `String` argument. This argument is passed to the method when the functional method `accept()` of the target type is invoked on the reference `outMR`, as shown at (2). The method `println()` is invoked on the same object (denoted by `System.out`) every time this method reference is executed. Execution of the lambda expression at (1a) will give the same result. Not surprisingly, a method reference and its corresponding single-method lambda expression are semantically equivalent.

Static Method References

Sometimes the lambda body of a lambda expression is just a call to a `static` *method*. The lambda expression at (1a) calls the `static` method `now()` in the class `java.time.LocalDate` to obtain the current date from the system clock.

Click here to view code image

```
Supplier<LocalDate> dateNowLE = () -> LocalDate.now();  // (1a) Lambda express
```

The lambda expression has the type `() -> LocalDate`, and not surprisingly, it is also the type of the static method `now()` which takes no arguments and returns an instance of the `LocalDate` class. The method type is compatible with the function type of the target type—that is, it is compatible with the type of the functional method `get()` of the parameterized functional interface `Supplier<LocalDate>`. The compiler can infer from the context that the type of the static method is compatible with the target type of the functional interface. In such a case, the lambda expression can be replaced by a *static method reference*:

Click here to view code image

```
Supplier<LocalDate> dateNowMR = LocalDate::now;         // (1b) Method referen
```

The double-colon delimiter (`::`) separates the reference type name (class `LocalDate`) from the static method name (`now`) in the syntax of the static method reference.

Analogous to a lambda expression, a method reference can be used as a value in an assignment, passed as an argument in a method or constructor call, returned in a `return` statement, or cast with the cast operator (p. 733). Its execution is deferred until the functional method of its target type is invoked, as at (2).

Click here to view code image

```
LocalDate today = datenowMR.get();    // (2) Method reference at (1b) executed
System.out.println(today.format(DateTimeFormatter.ISO_DATE)); // 2021-03-01
```

The following rule can be helpful in converting between a lambda expression and a static method reference:

> A lambda expression of the form

Click here to view code image

```
(args) -> RefType.staticMethod(args)
```

> is semantically equivalent to the *static method reference*:

Click here to view code image

```
RefType::staticMethod
```

where `RefType` is the *name* of a class, an enum type, or an interface that defines the static method whose *name* `staticMethod` is specified after the double-colon (`::`) delimiter.

Arguments are generally not specified in the method reference, and any parameters required by the method are determined by the target type of the context.

Click here to view code image

```
// String -> Integer
Function<String, Integer> strToIntLE = s -> Integer.parseInt(s); // (3a)
Function<String, Integer> strToIntMR = Integer::parseInt;        // (3b)
System.out.println(strToIntMR.apply("100"));                     // (4)
// Calls Integer.parseInt("100") that returns the int value 100 which is boxed
// into an Integer.
```

The `static` method `Integer.parseInt()` in the lambda body at (3a) takes one argument. Its type is `String -> Integer`. Using the one-arity `Function<String, Integer>` as the target type is appropriate as its function type is also `String -> Integer`. We can convert the lambda expression at (3a) to the static method reference shown at (3b). Note that no arguments are specified at (3b). The argument is passed to the static method at a later time when the functional method `apply()` of the functional interface is invoked, as demonstrated at (4).

Similarly, we can define static method references whose static method takes two arguments. A two-arity function or a binary operator from the `java.util.function` package can readily serve as the target type, as demonstrated by the following examples. Note that the static method `Math.min()` is overloaded, but the target type of the context determines which method will be executed. The type of the method `min()` at (5) is different than the type at (6). Analogous to lambda expressions, the same method reference can have different target types depending on the context.

Click here to view code image

```
// (double, double) -> double
DoubleBinaryOperator minDoubleLE = (x, y) -> Math.min(x, y);
DoubleBinaryOperator minDoubleMR = Math::min;          // (5)
System.out.println(minDoubleMR.applyAsDouble(20.0, 30.0));
// Calls Math.min(20.0, 30.0) that returns the double value 20.0.

// (int, int) -> (int)
IntBinaryOperator minIntLE = (x, y) -> Math.min(y, y);
IntBinaryOperator minIntMR = Math::min;                // (6)
System.out.println(minIntMR.applyAsInt(20, 30));
// Calls Math.min(20, 30) that returns the int value 20.
```

If the static method requires more that than two arguments, we can either define new functional interfaces with the appropriate arity for their functional method, or use the currying technique (p. 723).

Bounded Instance Method References

When the body of a lambda expression is a call to an *instance method*, the method reference specified depends on whether the object on which the instance method is invoked exists or not at the time the method reference is defined.

In the code below, the reference `sb` is declared and initialized at (1). It is effectively final when accessed in the lambda expression at (2a). The reference value of the reference `sb` is *captured* by the lambda expression. When the lambda expression is executed at a later time, the `reverse()` method is executed on the object denoted by the reference `sb`. In this case, the *bounded instance method reference* is simply the reference and the instance method name, separated by the double-colon delimiter, as shown at (2b). The bounded instance method reference at (2b) can replace the lambda expression at (2a). It is executed when the functional method `get()` of the parameterized functional interface `Supplier<StringBuilder>` is called, as shown at (3).

Click here to view code image

```
StringBuilder sb = new StringBuilder("!em esreveR");         // (1)
// () -> StringBuilder
Supplier<StringBuilder> sbReverserLE = () -> sb.reverse();    // (2a)
Supplier<StringBuilder> sbReverserMR = sb::reverse;           // (2b)
System.out.println(sbReverserMR.get());                      // (3)
// Calls sb.reverse() that returns the StringBuilder with character sequence
// "Reverse me!".
```

The case where the object on which the instance method is executed does not exist when the method reference is defined, but will be supplied when the method reference is executed, requires an *unbounded* instance method reference (p. 729).

The following rule can be used for converting between a lambda expression and a bounded instance method reference:

> A lambda expression of the form

Click here to view code image

```
(args) -> expr.instanceMethod(args)
```

> is semantically equal to the *bounded instance method reference*:

```
expr::instanceMethod
```

> where `expr` is an expression that evaluates to a reference value that is captured by the bounded instance method reference and becomes the *target reference* for the bounded instance method reference.

Any reference involved in the evaluation of `expr` must be effectively final, and is captured by the bounded instance method reference. The target reference represented by `expr` is separated from the instance method name by the double-colon delimiter. The target reference is fixed when the bounded instance method reference is defined. The instance method is invoked on the object denoted by the target reference when the method reference is executed at a later time, and any arguments required by the instance method are passed at the same time.

Given an `ArrayList<String>` denoted by the reference `words`, we can pass the method reference `System::println` to the `forEach()` method of the `ArrayList<E>` class in order to print each element of the list. The method takes a consumer as an argument, and the type of the object to print is inferred from the element type of the list.

Click here to view code image

```
words.forEach(obj -> System.out.println(obj));
words.forEach(System.out::println);
```

The syntax of the bounded instance method reference where the instance method has more than one argument is no different. In the code below, the `replace()` method of the `String` class has two arguments. Its type is `(String, String) -> String`, the same as the function type of the target type `BinaryOperator<String>`. The target reference is the reference `str` that is defined at (4). It is effectively final when accessed in the code, and its reference value is captured at (5b) where the method reference is defined. The method replaces each occurrence of `s1` in `str` with `s2`. The arguments are passed to the method when the functional method `apply()` of the target type `BinaryOperator<String>` is executed, as shown at (6).

Click here to view code image

```
String str = "Java Jive";                                              // (4)
// (String, String) -> String
BinaryOperator<String> replaceOpLE = (s1, s2) -> str.replace(s1, s2); // (5a)
BinaryOperator<String> replaceOpMR = str::replace;                     // (5b)
System.out.println(replaceOpMR.apply("Jive", "Jam"));                  // (6)
// Calls str.replace("Jive", "Jam") that returns the string "Java Jam".
```

In a *non-static* context, the `final` references `this` and `super` can also be used as the target reference of a bounded instance method reference.

Click here to view code image

```
Predicate<String> p1 = s -> this.equals(s);        // String -> boolean
Predicate<String> p2 = this::equals;               // String -> boolean
Supplier<String> s1 = () -> super.toString();      // () -> String
Supplier<String> s2 = super::toString;             // () -> String
```

Unbounded Instance Method References

In the case of an unbounded instance method reference, the target reference is determined when the method reference is executed, as it is the first argument passed to the method reference. This is embodied in the following rule:

> A lambda expression of the form

Click here to view code image

```
(arg0, rest) -> arg0.instanceMethod(rest)
```

> is semantically equivalent to the *unbounded instance method reference*:

```
RefType::instanceMethod
```

> where `RefType` is the reference type of the target reference `arg0`. The names of the reference type and the instance method are separated by the double-colon (`::`) delimiter.

The instance method is invoked on the object denoted by the target reference `arg0` (i.e., the first argument) when the method reference is executed, and any remaining arguments are passed to the instance method at the same time.

In the code below, the type of the unbounded instance method reference `String::length` at (1) is `String -> int`, the same as the function type of the target type `ToIntFunction<String>`. Invoking the functional method `applyAsInt()` on the reference `lenMR` results in the method `length()` being invoked on the string `"Java"` that was passed as a parameter.

Click here to view code image

```
// String -> int
ToIntFunction<String> lenLE = s -> s.length();
ToIntFunction<String> lenMR = String::length;                    // (1)
```

```
System.out.println(lenMR.applyAsInt("Java"));                    // 4
// Calls "Java".length() that returns the int value 4.
```

The `static` method `listBuilder()` in Example 13.6, p. 714, creates a list from an array by applying a `Function<T, R>` to each array element. An instance of the `Function<T, R>` is passed as a parameter to the method. Both lines of code below create a list of `Integer` from an array of `String`, where the functional interface parameter in the method call is inferred to be `Function<String, Integer>`. The method `length()` is executed on the first argument of the unbounded instance method reference— that is, on each `String` element of the list.

Click here to view code image

```
List<Integer> intList1 = listBuilder(strArray, s -> s.length()); // Lambda exp
List<Integer> intList2 = listBuilder(strArray, String::length);  // Method ref
```

The code below illustrates the case where the unbounded instance method reference `String::concat` at (2) requires two arguments. Its target type is `BinaryOperator<String>` that has the function type `(String, String) -> String`. The instance method `concat()` is invoked on the first argument, and the second argument is passed to the method as a parameter.

Click here to view code image

```
// (String, String) -> String
BinaryOperator<String> concatOpLE = (s1, s2) -> s1.concat(s2);
BinaryOperator<String> concatOpMR = String::concat;            // (2)
System.out.println(concatOpMR.apply("Java", " Jive"));         // Java Jive
// Calls "Java".concat(" Jive") that returns the string "Java Jive".
```

The code below illustrates using parameterized types in method references. At (3), the type of the argument of the generic interface `List<T>` in the method reference is inferred from the context to be `String`. At (4), the type of the argument is explicitly specified to be `String`. This can be necessary if the compiler cannot infer it from the context. The type of the `List::contains` instance method reference is `(List<String>, String) -> boolean`, which is compatible with the function type of the parameterized functional interface `BiPredicate<List<String>, String>`.

Click here to view code image

```
// (List<String>, String) -> boolean
BiPredicate<List<String>, String> containsLE
    = (list, element) -> list.contains(element);

BiPredicate<List<String>, String> containsMR1 = List::contains;          // (3
BiPredicate<List<String>, String> containsMR2 = List<String>::contains;  // (4
System.out.println(containsMR2.test(words, "BOB"));  // words is a List<String
// Calls words.contains("BOB") that returns a boolean value.
```

If the method in the unbounded instance method reference requires several arguments, compatible target types can be defined by either defining new functional interfaces with the appropriate arity for their functional method, or applying the currying technique (p. 723).

Constructor References

A constructor reference is similar to a static method reference, but with the keyword `new` instead of the static method name, signifying that a constructor of the specified class should be executed. Its purpose of course is to instantiate the class.

We can convert between a constructor reference and a lambda expression using the following rule:

> A lambda expression of the form

```
(args) -> new ClassType(args)
```

> is semantically equivalent to the *constructor reference*:

```
ClassType::new
```

> where `ClassType` is the name of the class that should be instantiated. The class name and the keyword `new` are separated by the double-colon (`::`) delimiter.
> Which constructor of `ClassType` is executed depends on the target type of the context, since it determines the arguments that are passed to the constructor.

Execution of the constructor reference `sbCR` defined at (1) results in the zero-argument constructor of the `StringBuilder` class to be executed, as evident from the type of the constructor reference.

Click here to view code image

```
// () -> StringBuilder
Supplier<StringBuilder> sbLE = () -> new StringBuilder();
Supplier<StringBuilder> sbCR = StringBuilder::new;                    // (1)
StringBuilder sbRef = sbCR.get();
// Calls new StringBuilder() to create an empty StringBuilder instance.
```

However, execution of the constructor reference `sb4` defined at (2) results in the constructor with the `String` parameter to be executed, as evident from the type of the constructor reference. The target types at (1) and (2) are different. The target type determines which constructor of the `StringBuilder` class is executed.

Click here to view code image

```
// String -> StringBuilder
Function<String, StringBuilder> sb3 = s -> new StringBuilder(s);
Function<String, StringBuilder> sb4 = StringBuilder::new;        // (2)
System.out.println(sb4.apply("Build me!"));                      // Build me!
// Calls new StringBuilder("Build me!") to create a StringBuilder instance
// based on the string "Build me!".
```

The following code illustrates passing two arguments to a constructor using an appropriate target type—in this case, a two-arity function—that ensures applicable arguments are passed to the constructor.

Click here to view code image

```
// (String, String) -> Locale
BiFunction<String, String, Locale> locConsLE
    = (language, country) -> new Locale(language, country);
BiFunction<String, String, Locale> locConsCR = Locale::new;
System.out.println(locConsCR.apply("en","US"));                  // en_US
// Calls new Locale("en", "US") to create a Locale instance with the specified
// parameter values.
```

Note that the constructor reference is *defined* first without any instance being created, and its execution is deferred until later when the functional method of its target type is invoked.

Array Constructor References

Array constructor reference is specialization of the constructor reference for creating arrays. We can convert between an array constructor reference and a lambda expression using the following rule:

> A lambda expression of the form

Click here to view code image

```
arg -> new ElementType[arg][]...[]
```

> is semantically equivalent to the *array constructor reference*:

```
ElementType[][]...[]::new
```

The array type and the keyword `new` are separated by the double-colon (`::`) delimiter. The `ElementType` is designated with the necessary pairs of square brackets (`[]`) to indicate that it is an array type of a specific number of dimensions. Only the length of the *first* dimension of the array can be created using an array constructor reference. As one would expect, the elements are initialized to the default value for the element type.

The array constructor reference at (1) will create a simple array of element type `int`. The target type `IntFunction<int[]>` is compatible with the type of the array constructor reference (`int -> int[]`). The array of `int` created at (2) has length 4, where each element has the default value `0`.

Click here to view code image

```
// int -> int[]
IntFunction<int[]> intArrConsLE = n -> new int[n];
IntFunction<int[]> intArrConsCR = int[]::new;                               //
int[] intArr = intArrConsCR.apply(4);                                       //
// Creates an int array of length 4.
```

In the code below, we can define a lambda expression to create a two-dimensional array that takes two arguments. However, this is *not* possible using an array constructor reference, as only the length of the first dimension can be passed to an array constructor reference. The line at (3) will not compile, since the target type (`(Integer, Integer) -> int[][]`) is not compatible with the type of the array constructor reference (`int -> int[][]`).

Click here to view code image

```
// (int, int) -> int[][]
BiFunction<Integer, Integer, int[][]> twoDimArrConsLE1 = (n, m) -> new int[n][
// BiFunction<Integer, Integer, int[][]> twoDimArrConsCR1
//      = int[][]:: new;                               // (3) Compile-time err
```

It is only possible to define an array constructor reference to create the length of the first dimension of an array, regardless of how many dimensions it has. This is illustrated by the array constructor reference at (4), which creates a two-dimensional array where only the first dimension is constructed—keeping in mind that in Java, multidimensional arrays are implemented as arrays of arrays. The code at (5) returns an array with three rows, where each row is initialized to the `null` value. Individual arrays can be constructed and stored in the two-dimensional array, as shown at (6).

Click here to view code image

```
// int -> int[][]
IntFunction<int[][]> twoDimArrConsLE = n -> new int[n][];
IntFunction<int[][]> twoDimArrConsCR = int[][]::new;            // (4)
```

```
int[][] twoDimIntArr1 = twoDimArrConsCR.apply(3);               // (5)
// [null, null, null]
for (int i = 0; i < twoDimIntArr1.length; ++i)
  twoDimIntArr1[i] = intArrConsCR.apply(i+1);                   // (6) Calls (
// [[0], [0, 0], [0, 0, 0]]
```

The example below illustrates constructing an array of objects. The procedure is no different from constructing arrays of primitive values, as explained above. The array returned by the code at (7) has five elements, where each element is initialized to the `null` value.

Click here to view code image

```
// int -> StringBuilder[]
IntFunction<StringBuilder[]> sbaConsLE = n -> new StringBuilder[n];
IntFunction<StringBuilder[]> sbaConsCR = StringBuilder[]::new;
StringBuilder[] sbArr2 = sbaConsCR.apply(5);                      // (7)
// [null, null, null, null, null]
```

Java does not allow creation of generic arrays, as demonstrated by the declaration statement at (8), where an attempt is made to construct an array of formal parameter type `A` (§11.13, p. 627). This can be overcome by using either a lambda expression or an array constructor reference whose target type is a parameterization of `IntFunction<A[]>`, and which is passed to the generic method `createArray()` below, together with the required array length, to create an array of a specific type.

Click here to view code image

```
public static <A> A[] createArray(int length, IntFunction<A[]> creator) {
//A[] arr = new A[length];       // (8) Cannot create generic array!
  return creator.apply(length);  // Lambda expression or

                                 // array constructor reference executed.
}
```

The code below calls the generic method `createArray()` with a lambda expression and an array constructor reference at (9) and (10), respectively, to create a `String` array of length 5. The target type in both cases is parameterized functional interface `IntFunction<String[]>`.

Click here to view code image

```
// n -> String[]
String[] strArrLE = createArray(5, n -> new String[n]); // (9)
String[] strArrACE = createArray(5, String[]::new);     // (10)
```

13.14 Contexts for Defining Lambda

Expressions

In this section, we summarize the main contexts that can provide target types for lambda expressions and method references.

Declaration and Assignment Statements

Ample examples of defining lambda expressions and method references in this context have been presented throughout this chapter. As we have seen earlier, the target type is inferred from the type of the assignment target—that is, the functional interface type being assigned to on the left-hand side of the assignment operator.

Click here to view code image

```
DoubleFunction cToF = x -> 1.8 * x + 32.0;           // double -> double
ToIntFunction<String> lenFunc1 = s -> s.length();    // String -> int
ToIntFunction<String> lenFunc2 = String::length;     // String -> int
lenFunc1 = s -> Integer.parseInt(s);                 // String -> int
lenFunc2 = Integer::parseInt;                        // String -> int
```

Method and Constructor Calls

We have seen several examples where a lambda expression is passed as an argument in a method or constructor call. The target type is the functional interface type of the corresponding formal parameter.

Click here to view code image

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.forEach(i -> System.out.println(i));   // Target type: Consumer<Intege
numbers.forEach(System.out::println);
```

Expressions in `return` Statements

The expression in a `return` statement can define a lambda expression (or a method reference) whose target type is the return type of the method. The method below returns a function of type `int -> int`:

Click here to view code image

```
static IntUnaryOperator createLinearFormula(int a, int b) {
  return x -> a * x + b;      // int -> int
}

// Client code:
IntUnaryOperator y = createLinearFormula(10, 5);  // 10 * x + 5
```

```
y.applyAsInt(2);                                         // 25
```

Ternary Conditional Expressions

For lambda expressions defined in a ternary conditional expression, the target type is inferred from the context of the ternary conditional expression.

In the first ternary conditional expression below, the target type for the lambda expressions that are operands is inferred from the target of the assignment statement, which happens to be an `IntUnaryOperator` interface.

Click here to view code image

```
int ii = 10;
IntUnaryOperator iFunc1 = ii % 2 == 0 ? i -> i * 2 : j -> j + 1;   // int -> i
iFunc1.applyAsInt(4);                                              // 8
//IntUnaryOperator iFunc2 = ii % 2 == 0 ? i -> i * 2
//                          : s -> Integer.parseInt(s);   // Compile-time erro
```

In the second ternary conditional expression above, the code does not compile because the type `String -> int` of the lambda expression that is the second operand is not compatible with the type `int -> int` of the assignment target.

Cast Expressions

The context of a cast expression can provide the target type for a lambda expression or a method reference.

In the two statements below, the cast provides the target type of the lambda expression. Note that the textual lambda expressions are identical, but their target types are different.

Click here to view code image

```
System.out.println(((IntUnaryOperator) i -> i * 2).applyAsInt(10));         //
System.out.println(((DoubleUnaryOperator) i -> i * 2).applyAsDouble(10.0));//
```

The first statement below does not compile, as the `Object` class is not a functional interface and therefore cannot provide a target type. In the second statement, the cast provides the target type of the constructor expression. The type `Function<String, StringBuilder>` is a subtype of the supertype `Object`, and the assignment is allowed.

Click here to view code image

```
// Object obj1 = StringBuilder::new;                 // Compile-time error!
Object obj2 = (Function<String, StringBuilder>) StringBuilder::new;
```

In the code below, the `instanceof` operator at (1) is used to guarantee that at runtime the cast will succeed at (2) and the lambda expression can be executed at (3). Without the `instanceof` operator, the cast at (2) will be allowed at compile time, but a resounding `ClassCastException` will be thrown at runtime, as `DoubleUnaryOperator` and `IntUnaryOperator` are not related types. In the code below, the body of the `if` statement is not executed.

Click here to view code image

```
Object uFunc1 = (IntUnaryOperator) i -> i * 2;
if (uFunc1 instanceof DoubleUnaryOperator) {                      // (1) false
  DoubleUnaryOperator uFunc2 = (DoubleUnaryOperator) uFunc1;      // (2)
  uFunc2.applyAsDouble(10.0);                                     // (3)
}
```

Nested Lambda Expressions

When lambda expressions are nested, the context of the outer lambda expressions can provide the target type for the inner lambda expressions. This typically occurs when currying functions.

Click here to view code image

```
Supplier<Supplier<String>> f = () -> () -> "Hi";
```

The target type for the nested lambda expressions is inferred from the context, which is an assignment statement to a reference of type `Supplier<Supplier<String>>`. The inner lambda expression `() -> "Hi"` is inferred to be of target type `Supplier<String>`, as its type `() -> String` is compatible with the function type of this target type. The outer lambda expression is then inferred to have the type `() -> Supplier<String>` which is compatible with the target type `Supplier<Supplier<String>>`.

Review Questions

13.4 Given the following code:

Click here to view code image

```
      import java.util.*;
```

```
public class Test13RQ5 {
  public static void main(String[] args) {
    List<String> values = new ArrayList<>(List.of("ONE","TWO","THREE","F
    values.removeIf(s -> s.length() == 3);
    int sum = 0;
    for (String value: values) {
      sum += value.length();
    }
    System.out.println(sum);
  }
}
```

What is the result?
Select the one correct answer.

a. `3`
b. `6`
c. `9`
d. The program will throw an exception at runtime.
e. The program will fail to compile.

13.5 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13RQ6 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    values.removeIf(s -> s.toLowerCase().startsWith("a"));
    System.out.println(values);
  }
}
```

What is the result?
Select the one correct answer.

a. `[jane, john]`
b. `[anna, alice]`
c. `[JANE, JOHN]`
d. `[ANNA, ALICE]`
e. `[ANNA, JANE, ALICE, JOHN]`
f. `[anna, jane, alice, john]`
g. The program will compile, but it will not produce any output when run.
h. The program will throw an exception at runtime.
i. The program will fail to compile.

13.6 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13RQ8 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    String s = values.get(0).substring(0,1);
    values.removeIf(s -> s.toLowerCase().startsWith(s));
    values.forEach(s -> System.out.print(s + " "));
  }
}
```

What is the result?
Select the one correct answer.

a. `jane john`
b. `anna alice`
c. `JANE JOHN`
d. `ANNA ALICE`
e. `ANNA JANE ALICE JOHN`
f. `anna jane alice john`
g. The program will compile, but it will not produce any output when run.
h. The program will throw an exception at runtime.
i. The program will fail to compile.

13.7 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ9 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("PLOT","FLOP","LOOP","LEAP"));
    Predicate<String> filter1 = s -> s.contains("O");
    Predicate<String> filter2 = s -> s.endsWith("P");
    values.removeIf(filter1.and(filter2).negate());
    System.out.println(values);
  }
}
```

What is the result?
Select the one correct answer.

a. `[LEAP]`
b. `[PLOT]`
c. `[FLOP, LOOP]`
d. `[PLOT, FLOP, LOOP]`
e. `[PLOT, FLOP, LOOP, LEAP]`
f. The program will compile, but it will not produce any output when run.
g. The program will throw an exception at runtime.
h. The program will fail to compile.

13.8 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ10 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
    UnaryOperator<String> f1 = v1 -> v1.substring(0,1).toUpperCase();
    UnaryOperator<String> f2 = v2 -> v2.substring(1).toLowerCase();
    UnaryOperator<String> f3 = f1.compose(f2);
    values.replaceAll(f3);
    System.out.println(values);
  }
}
```

What is the result?
Select the one correct answer.

a. `[Alice, Bob, John, Jane]`

b. `[aLICE, bOB, jOHN, jANE]`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

13.9 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ11 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
    UnaryOperator<String> f1 = v -> v.toLowerCase();
    values.replaceAll(f1);
    Consumer<String> c1 = s -> s = s.substring(0,1).toUpperCase();
    Consumer<String> c2 = s -> System.out.print(s + " ");
    values.forEach(c1.andThen(c2));
  }
}
```

What is the result?
Select the one correct answer.

a. `Alice Bob John Jane`

b. `alice bob john jane`

c. `A B J J`

d. The program will throw an exception at runtime.

e. The program will fail to compile.

13.10 Which method references are equivalent to lambda expressions at (1) and (2) in the following code?

Click here to view code image

```
import java.util.*;
class Test {
  private List<Integer> values = new ArrayList<>(List.of(1,2,3,4,5));
  public List<Integer> getValues()         { return values; }
  public static boolean isEven(int value)  { return value % 2 != 0; }
  public void printValue(int value)        { System.out.print(value + " "
}

public class Main {
  public static void main(String[] args) {
    Test test = new Test();
    test.getValues().removeIf(v -> v % 2 != 0);               // (1)
    test.getValues().forEach(v -> System.out.print(v + " ") ); // (2)
  }
}
```

Select the one correct answer.

a. `test.getValues().removeIf(Test::isEven);`
`test.getValues().forEach(Test::printValue);`

b. `test.getValues().removeIf(Test::isEven);`
`test.getValues().forEach(test::printValue);`

c. `test.getValues().removeIf(test::isEven);`
`test.getValues().forEach(test::printValue);`

d. `test.getValues().removeIf(test::isEven);`
`test.getValues().forEach(Test::printValue);`

e. None of the above

13.11 Which statement is true about method referencing?

a. Unbounded instance method reference determines the target reference as the first argument passed to the method reference.

b. Unbounded instance method reference determines the target reference as the last argument passed to the method reference.

c. Bounded instance method reference determines the target reference as the first argument passed to the method reference.

d. Bounded instance method reference determines the target reference as the last argument passed to the method reference.

13.12 Given the following code:

Click here to view code image

```
import java.util.function.BiFunction;
public class Test24RQ6 {
  public static void main(String[] args) {
    BiFunction divide = (x, y) -> x/y;
    System.out.print(divide.apply(0.0,0));
  }
}
```

What is the result?
Select the one correct answer.

a. `0`

b. `0.0`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

13.13 Given the following code:

Click here to view code image

```
import java.util.function.Function;
public class Test13RQ19 {
  public static void main(String[] args) {
    Function f1 = (x) -> "<" + x;
    Function f2 = (x) -> x + ">";
    System.out.print(f2.compose(f1).apply(42));
  }
}
```

What is the result?
Select the one correct answer.

a. `<42>`
b. `>42<`
c. The program will throw an exception at runtime.
d. The program will fail to compile.

Object Comparison 14

Chapter Topics

- Overview of selected convenience methods of the `Objects` class
- How to correctly override and implement the contract of the `equals()` method of the `Object` class
- How to correctly override and implement the contract of the `hash-Code()` method of the `Object` class
- Comparing objects whose natural order is defined by the contract of the `compareTo()` method of the `java.lang.Comparable<T>` interface
- Comparing objects using a comparator that defines a total order according to the contract of the `compare()` method of the `java.util.Comparator<T>` interface

Java SE 17 Developer Exam O	
§14.1, p. 743 §14.2, p. 744 §14.3, p. 753	[3.5] Implement inheritance, including abstract and sealed classes. Override methods, including that of Object class. Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching ❍ *Overriding methods of the* `Object` *class is covered in this chapter.*

- *For inheritance, abstract and sealed classes, overriding methods, polymorphism and type casting, static and dynamic type of a reference,* `instanceof` *type comparison and pattern match operators, see Chapter 5, p. 189.*

[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements

§14.4, p. 761
§14.5, p. 769

- `Comparable` *and* `Comparator` *interfaces for comparing objects are covered in this chapter.*
- *For arrays, see §3.9, p. 117.*
- *For* `ArrayList`*, see Chapter 12, p. 643.*
- *For lists, sets, maps, deques, and sorted collections and arrays, see Chapter 15, p. 781.*

Java SE 11 Developer Exam O

[5.3] Sort collections and arrays using Comparator and Comparable interfaces

§14.4, p. 761
§14.5, p. 769

- `Comparable` *and* `Comparator` *interfaces are covered in this chapter.*
- *For sorted collections and arrays, see Chapter 15, p. 781.*

This chapter covers the important topic of comparing objects. Many useful operations require that the objects can be compared for *object value equality* and *ranked* according to some meaningful criteria. Typical examples are searching and sorting algorithms for objects maintained in arrays, collections, and maps. Overriding the methods `equals()` and `hashCode()` in the `Object` class and the methods `compareTo()` and `compare()` in the interfaces `Comparable<T>` and `Comparator<T>`, respectively, is essential in this regard.

The `equals()` and `hashCode()` methods of the `Object` class provide specific contracts for objects, which the classes overriding the methods should honor. It is important to understand how and why a class should override the `equals()` and `hashCode()` methods.

The `compareTo()` method of the `Comparable<E>` interface defines the contract for comparing objects which classes can implement. In addition, the `compare()` method of the `Comparator<E>` interface allows other criteria to be defined for comparing objects.

Objects of a class that override the `equals()` method make it possible to search for such objects in arrays and lists. If they override the `hashCode()` method, they can also be searched for as elements in a set and as keys in a map. Implementing the `Comparable<E>` interface allows them to be sorted and maintained as elements in sorted collections and as keys in sorted maps. Collections and maps are covered in detail in Chapter 15, p. 781.

14.1 The `Objects` Class

The following `static` methods of the `java.util.Objects` class are convenient to use when comparing objects and computing the hash code for an object. They take away the drudgery of checking references for `null` values before calling them and never throw a `NullPointerException`—that is, they are `null`-safe.

> Click here to view code image
>
> ```
> static boolean equals(Object obj1, Object obj2)
> ```
>
> Returns `true` if the arguments are equal to each other; otherwise, it returns `false`. This means that if both arguments are `null`, `true` is returned and if only one argument is `null`, `false` is returned. Equality is determined by invoking the `equals()` method on the first argument: `obj1.equals(obj2)`. It is a convenience method for the `Object.equals()` method (Example 14.3, p. 748).
>
> Click here to view code image
>
> ```
> static int hash(Object... values)
> static int hashCode(Object obj)
> ```
>
> The first method generates a hash code for a sequence of specified values. It is useful for computing the hash code for objects containing multiple fields that are passed as arguments (Example 14.6, p. 771).
>
> The second method computes and returns the hash code of a non-`null` argument; otherwise, it returns `0` for a `null` argument.
>
> Both methods are convenience methods for the `Object.hashCode()` method.
>
> Click here to view code image
>
> ```
> static <T> int compare(T t1, T t2, Comparator<? super T> cmp)
> ```
>
> Returns `0` if the arguments are identical; otherwise, it calls `cmp.compare(t1, t2)`, which is the abstract method in the `Comparator<T>` interface. It also returns `0` if both arguments are `null` (Example 14.8, p. 777).

14.2 Implementing the `equals()` Method

If every object is to be considered unique, then it is not necessary to override the `equals()` method in the `Object` class. This method implements *object reference equality*. It implements the most discriminating equivalence relation possible on objects. Each instance of the class is only equal to itself.

> Click here to view code image
>
> ```
> public boolean equals(Object obj)
> ```
>
> The `equals()` method in the `Object` class tests for *object reference equality*, the same as the `==` operator. It returns `true` only if the two references compared denote the same object—that is, if they are aliases. The `equals()` method is usually overridden to provide the semantics of *object value equality*, as in the case of the wrapper classes and the `String` class.

As a running example, we will implement different versions of a class for *version numbers*. A version number (VNO) for a software product comprises three pieces of information:

- A release number
- A revision number
- A patch number

The idea is that releases do not happen very often. Revisions take place more frequently than releases, but less frequently than code patches are issued. We can say that the release number is most *significant*. The revision number is less significant than the release number, and the patch number is the least significant of the three fields. This ranking would also be employed when ordering version numbers chronologically.

Example 14.1 *Not Overriding the* `Object.equals()` *Method*

Click here to view code image

```
/** A simple version number class */
public class SimpleVNO {

  private int release;
  private int revision;
  private int patch;

  public SimpleVNO(int release, int revision, int patch) {
    this.release  = release;
```

```
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }
}
```

The class `SimpleVNO` in Example 14.1 does not override the `equals()` method in the `Object` class. It only overrides the `toString()` method to generate a meaningful text representation for a version number.

The class `TestSimpleVNO` in Example 14.2 creates objects of the class `SimpleVNO` in Example 14.1 to test for *object reference equality* and *object value equality*. The output from Example 14.2 demonstrates that all `SimpleVNO` objects are unique because the class `SimpleVNO` does not override the `equals()` method to provide any other equivalence relation. The result of the == operator is always `false` for object reference equality, since all `SimpleVNO` objects are unique. An attempt is made to compare `SimpleVNO` objects for object value equality using the `Object.equals()` method, but the implementation of the `Object.equals()` method tests for object reference equality, the same as the == operator. Not surprisingly, the `equals()` method returns `false` even for `SimpleVNO` objects that have the same state (`svno1` and `svno2`).

Example 14.2 *Implications of Not Overriding the* `Object.equals()` *Method*

Click here to view code image

```
import static java.lang.System.out;

public class TestSimpleVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(SimpleVNO.class);

    // Three individual version numbers.
    SimpleVNO svno1 = new SimpleVNO(9,1,1);                         // (1)
    SimpleVNO svno2 = new SimpleVNO(9,1,1);                         // (2)
    SimpleVNO svno3 = new SimpleVNO(6,6,6);                         // (3)

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
    out.println("Test object reference equality:");                 // (4)
    out.println("  svno1 == svno2:       " + (svno1 == svno2));     // (5)
    out.println("  svno1 == svno3:       " + (svno1 == svno3));     // (6)
    out.println("Test object value equality:");
```

```
      out.println("  svno1.equals(svno2): " + svno1.equals(svno2));  // (7)
      out.println("  svno1.equals(svno3): " + svno1.equals(svno3));  // (8)
    }
  }
```

Output from the program:

Click here to view code image

```
class SimpleVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): false
  svno1.equals(svno3): false
```

Equivalence Relation of the `equals()` Method

An implementation of the `equals()` method must satisfy the properties of an *equivalence relation*:

- *Reflexive*: For any reference `self`, `self.equals(self)` is always `true`.
- *Symmetric*: For any references `x` and `y`, if `x.equals(y)` is `true`, then `y.equals(x)` is `true`.
- *Transitive*: For any references `x`, `y`, and `z`, if both `x.equals(y)` and `y.equals(z)` are `true`, then `x.equals(z)` is `true`.
- *Consistent*: For any references `x` and `y`, multiple invocations of `x.equals(y)` will always return the same result, provided the objects referenced by these references have not been modified to affect the equals comparison.
- `null` *comparison*: For any non-null reference `obj`, the call `obj.equals(null)` always returns `false`.

The general contract of the `equals()` method is defined between *objects of arbitrary classes*. Understanding its criteria is important for providing a proper implementation.

Reflexivity

This rule simply states that an object is equal to itself, regardless of how it is modified. It is easy to satisfy: The object passed as an argument and the current object are compared for object reference equality (==), the same as the default behavior of the

`Object.equals()` method. If the references are aliases, the `equals()` implementation returns `true`.

```
if (this == argumentObj)
  return true;
// ...
```

The reflexivity test can usually be an operand of the short-circuit conditional-OR operator (`||`) in a `return` statement, where if the reflexivity test is `true`, other criteria for equality comparison are not evaluated:

Click here to view code image

```
return (this == argumentObj)
    || (/* Other criteria for equality. */);
```

Symmetry

The expression `x.equals(y)` invokes the `equals()` method on the object referenced by the reference `x`, whereas the expression `y.equals(x)` invokes the `equals()` method on the object referenced by the reference `y`. If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.

If the `equals()` methods invoked are in different classes, the classes must bilaterally agree whether their objects are equal or not. In other words, symmetry can be violated if the `equals()` method of a class makes unilateral decisions about which classes it will interoperate with, while the other classes are not aware of this. Avoiding interoperability with other (non-related) classes when implementing the `equals()` method is strongly recommended.

Transitivity

If two classes, `A` and `B`, have a bilateral agreement on their objects being equal, then this rule guarantees that one of them, say `B`, does not enter into an agreement with a third class `C` on its own. All classes involved must multilaterally abide by the terms of the contract.

A typical pitfall resulting in broken transitivity is when the `equals()` method in a subclass calls the `equals()` method of its superclass, as part of its equals comparison. The `equals()` method in the subclass can be implemented by code equivalent to the following line:

Click here to view code image

```
return super.equals(argumentObj) && compareSubclassSpecificAspects();
```

The idea is to compare only the subclass-specific aspects in the subclass `equals()` method and to use the superclass `equals()` method for comparing the superclass-specific aspects. However, this approach should be used with extreme caution. The problem lies in getting the equivalence contract fulfilled bilaterally between the superclass and the subclass `equals()` methods. Symmetry or transitivity can easily be broken.

If the superclass is abstract, using the superclass `equals()` method works well. There are no superclass objects for the subclass `equals()` method to consider. In addition, the superclass `equals()` method cannot be called directly by any clients other than subclasses. The subclass `equals()` method then has control of how the superclass `equals()` method is called. It can safely call the superclass `equals()` method to compare the superclass-specific aspects of subclass objects.

Consistency

This rule enforces that two objects that are equal (or non-equal) remain equal (or non-equal) as long as they are not modified. For mutable objects, the result of the equals comparison can change if one (or both) are modified between method invocations. However, for immutable objects, the result must always be the same. The `equals()` method should take into consideration whether the class implements immutable objects, and ensure that the consistency rule is not violated.

`null` *Comparison* This rule states that no object is equal to `null`. The contract calls for the `equals()` method to return `false`. The method must not throw an exception; that would be violating the contract. A check for this rule is necessary in the implementation. Typically, the argument object is explicitly compared with the `null` value:

```
if (argumentObj == null)
  return false;
```

In many cases, it is preferable to use the `instanceof` pattern match operator. It determines whether the argument object is of the appropriate type and introduces a local variable of the appropriate subtype that denotes the argument object. It is always `false` if its left operand is `null`:

Click here to view code image

```
if (!(argumentObj instanceof MyRefType other))
  return false;
// Local variable other can be used in further implementation.
```

Note that if the `instanceof` pattern match operator is `true`, the `if` condition is `false` so that the local variable `other` of `MyRefType` is introduced into the scope *after* the `if` statement. In fact, the `if` statement can often be eliminated if the fields are compared individually:

Click here to view code image

```
return (this == argumentObj)                          // Reflexivity test
    || (argumentObj instanceof MyRefType other        // null comparison & cas
        && /* Can use variable other to compare individual fields. */);
```

The `return` statement above will now also return `false` if the `instanceof` pattern match operator determines that `argumentObj` is `null` or if it is not of `MyRefType`. If the `instanceof` pattern match operator returns `true`, the reference `other` can be safely used to compare the individual fields by short-circuit evaluation of the `&&` operator (see Example 14.3).

Example 14.3 *Implementing the* `equals()` *Method*

Click here to view code image

```
import java.util.Objects;

// Overrides equals(), but not hashCode().
public class UsableVNO {

  private int release;
  private int revision;
  private int patch;

  public UsableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {     // (1)
    return (this == obj)                            // (2)
        || (obj instanceof UsableVNO vno            // (3)
            && this.patch    == vno.patch           // (4)
            && this.revision == vno.revision
            && this.release  == vno.release);
  }
}
```

Checklist for Implementing the `equals()` Method

Example 14.3 shows an implementation of the `equals()` method for version numbers. Next, we provide a checklist for implementing the `equals()` method.

Method Overriding Signature

The method header is

Click here to view code image

```
public boolean equals(Object obj)          // (1)
```

The signature of the method requires that the argument passed is of the type `Object`. The following header will overload the method, not override it:

Click here to view code image

```
public boolean equals(MyRefType obj)      // Overloaded.
```

The compiler will not complain. Therefore, it is a good idea to use the `@Override` annotation. Calls to overloaded methods are resolved at compile time, depending on the type of the argument. Calls to overridden methods are resolved at runtime, depending on the type of the actual object referenced by the argument. Comparing the objects of the class `MyRefType` that *overloads* the `equals()` method for equivalence can give inconsistent results:

Click here to view code image

```
MyRefType ref1 = new MyRefType();
MyRefType ref2 = new MyRefType();
Object    ref3 = ref2;
boolean b1 = ref1.equals(ref2);    // True. Calls equals() in MyRefType.
boolean b2 = ref1.equals(ref3);    // Always false. Calls equals() in Object.
```

However, if the `equals()` method is overridden correctly, only the overriding method in `MyRefType` is called. A class can provide both implementations, but the `equals()` methods must be consistent. However, there must be a legitimate reason to overload the `equals()` method and this practice is not recommended.

Reflexivity Test

This is usually the first test performed in the `equals()` method, avoiding further computation if the test is `true`. The `equals()`

method in Example 14.3 does this test at (2):

Click here to view code image

```
return (this == obj)                                // (2)
    || (/* */);
```

Correct Argument Type

The `equals()` method of the `UsableVNO` class in Example 14.3 checks the type of the argument object and assigns its reference value to the local variable `vno` at (3) using the `instanceof` pattern match operator:

Click here to view code image

```
return (this == obj)                                // (2)
    || (obj instanceof UsableVNO vno                // (3)
         && /* Compare individual fields */);
```

This code also does the `null` comparison correctly, returning `false` if the argument `obj` has the value `null`.

The `instanceof` operator will also return `true` if the argument `obj` denotes a *subclass object* of the class `UsableVNO`. If the class is `final`, this issue does not arise—there are no subclass objects. The following test can be explicitly performed in order to exclude all other objects, including subclass objects:

Click here to view code image

```
if ((obj == null) || (obj.getClass() != this.getClass()))
  return false;
```

The condition in the `if` statement first performs the `null` comparison. The expression `(obj.getClass() != this.getClass())` determines whether the classes of the two objects have the same runtime object representing them. If this is the case, the objects are instances of the same class.

Field Comparisons

Equivalence comparison involves comparing relevant fields from both objects to determine if their logical states match. For fields that are of primitive data types, their primitive values can be compared. Instances of the class `UsableVNO` in Example 14.3 have fields of primitive data types only. Values of corresponding fields can be compared to test for equality between two `UsableVNO` objects, as shown at (4):

Click here to view code image

```
return (this == obj)                                 // (2)
    || (obj instanceof UsableVNO vno                 // (3)
        && this.patch    == vno.patch                // (4)
        && this.revision == vno.revision
        && this.release  == vno.release);
```

If all field comparisons evaluate to `true`, the `equals()` method returns `true`.

For fields that are references, the objects referenced by the references can be compared. For example, if the `UsableVNO` class declares a field called `productInfo`, which is a reference, the following expression can be used:

Click here to view code image

```
(this.productInfo != null && this.productInfo.equals(vno.productInfo)))
```

In order to avoid a `NullPointerException` being thrown, the `equals()` method is not invoked if the `this.productInfo` reference is `null`. However, the `Objects.equals()` utility method makes this comparison much simpler, as it provides `null`-safe object value equality operation, only invoking the `equals()` method on the first `productInfo` field, if both arguments are not `null`:

Click here to view code image

```
Objects.equals(this.productInfo, vno.productInfo)
```

Exact comparison of floating-point values should not be done directly on the values, but on the integer values obtained from their bit patterns (see static methods `Float.floatToIntBits()` and `Double.doubleToLongBits()` in the Java SE Platform API documentation). This technique eliminates certain anomalies in floating-point comparisons that involve a NaN value or a negative zero (see also the `equals()` method in `Float` and `Double` classes).

Only fields that have significance for the equivalence relation should be considered. Derived fields, whose computation is dependent on other field values in the object, might be redundant to include. Including only the significant fields may be prudent. Computing the equivalence relation should be deterministic; therefore, the `equals()` method should not depend on unreliable resources, such as network access.

The order in which the comparisons of the significant fields are carried out can influence the performance of the equals comparison. Fields that are most likely to differ should be compared as early as possible, in order to short-circuit the computation. In our example, patch numbers evolve faster than revision numbers, which, in turn, evolve faster than release numbers. This order is reflected in the `return` statement at (4) in Example 14.3.

Above all, an implementation of the `equals()` method must ensure that the equivalence relation is fulfilled.

Example 14.4 is a client that uses the class `UsableVNO` from Example 14.3. This client runs the same tests as the client in Example 14.2. The difference is that the class `UsableVNO` overrides the `equals()` method.

Example 14.4 *Implications of Implementing the* `equals()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(UsableVNO.class);

    // Three individual version numbers.
    UsableVNO svno1 = new UsableVNO(9,1,1);                               // (
    UsableVNO svno2 = new UsableVNO(9,1,1);                               // (
    UsableVNO svno3 = new UsableVNO(6,6,6);                               // (

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                             // (
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
    out.println("Test object reference equality:");                       // (
    out.println("  svno1 == svno2:       " + (svno1 == svno2));          // (
    out.println("  svno1 == svno3:       " + (svno1 == svno3));          // (
    out.println("Test object value equality:");
    out.println("  svno1.equals(svno2): " + svno1.equals(svno2));       // (
    out.println("  svno1.equals(svno3): " + svno1.equals(svno3));       // (
    out.println();

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                           // (

    // Searching in an array:
    boolean found = false;
    for (UsableVNO version : versions) {
      found = searchKey.equals(version);                                  // (
      if (found) break;
    }
    out.println("Array: " + Arrays.toString(versions));                    // (
    out.printf("  Search key %s found in array:    %s%n%n",                // (
                searchKey, found);

    // Searching in a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                      // (
    out.println("List:  " + vnoList);
    out.printf("  Search key %s contained in list: %s%n%n", searchKey,
                vnoList.contains(searchKey));                               // (
  }
}
```

Output from the program:

Click here to view code image

```
class UsableVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): true
  svno1.equals(svno3): false

Array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) found in array:    true

List:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) contained in list: true
```

The output from the program shows that object value equality is compared correctly. Object value equality is now based on identical states, as defined by the `equals()` method.

The search for a `UsableVNO` object in an array or a list of `UsableVNO` objects is successful, since the equals comparison is based on the states of the objects and not on their reference values.

Next, we look at how to fix the version numbers so that they can be used for searching in sets and maps.

14.3 Implementing the `hashCode()` Method

Hashing is an efficient technique for storing and retrieving data. A common hashing scheme uses an array where each element is a list of items. The array elements are called *buckets*. Operations in a hashing scheme involve computing an array index from an item. Converting an item to its array index is done by a *hash function*. The array index returned by the hash function is called the *hash code* of the item— it is also called the *hash value* of the item. The hash code identifies a particular bucket.

Storing an item involves the following steps:

- Hash the item to determine the bucket.
- If the item does not match an item already in the bucket, it is stored in the bucket.

Note that no duplicate items are stored. A lookup for an item is also a two-step process:

- Hash the item to determine the bucket.
- If the item matches an item in the bucket, the item is present; otherwise, it is not.

Different items can hash to the same bucket, meaning that the hash function returns the same hash code for these items. This condition is called a *collision*. The list maintained by a bucket contains the items that hash to the bucket.

The hash code of an item only identifies the bucket. Finding an item in the bucket entails a search, and requires an equality function to compare items. The items maintained in a hash-based storage scheme must, therefore, provide two essential functions: a hash function and an equality function.

The performance of a hashing scheme is largely affected by how well the hash function distributes a collection of items over the available buckets. A hash function should not be biased toward any particular hash codes. An ideal hash function produces a uniform distribution of hash codes for a collection of items across all possible hash codes. Such a hash function is not an easy task to design. Fortunately, heuristics exist for constructing adequate hash functions.

Here we mention two abstract data types that can be implemented using a hashing scheme:

- A *set* is an abstract data type that stores an unordered collection of unique items. Items in a set are called *elements* and the element to search for in a set is referred to as the *search key*. The hashing is used to store and lookup an item in a set, as explained earlier in this section.
- A *map* (also called, a *hash table*) is an abstract data type that maintains its items as *key–value entries*. An *entry* associates a *key* with a *value*. The keys in a hash table are

unique. The hashing is done on a search key to provide efficient lookup of the entry with the associated value. Matching the search key with a key in an entry determines the value.

If objects of a class are to be maintained in hash-based collections and maps provided by the `java.util` package (see Table 15.2, p. 788), the class must provide appropriate implementations of the following methods from the `Object` class:

- A `hashCode()` method that produces hash codes for the objects
- An `equals()` method that tests objects for object value equality

> `int hashCode()`
>
> When storing objects in hash tables, this method can be used to get a hash code for an object. This method tries to return distinct integers for distinct objects as their default hash code. The `hashCode()` method is usually overridden by a class, as in the case of the wrapper classes and the `String` class.

As a general rule for implementing these methods, *a class that overrides the* `equals()` *method must override the* `hashCode()` *method.* Consequences of not doing so are illustrated in Example 14.5 using the class `UsableVNO` that does not override the `hash-Code()` method. Elements of this class are used as elements in a set that uses the `hashCode()` method of its elements to maintain them in the set. The output from the program shows that a set with the following elements is created:

Click here to view code image

```
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
```

The `hashCode()` method from the `Object` class is not overridden by the `UsableVNO` class and is, therefore, used to compute the hash codes of the elements. This method returns the memory address of the object as the default hash code. The attempt to find the search key `(9.1.1)` in the set is unsuccessful:

Click here to view code image

```
Search key (9.1.1) contained in set: false
```

The output from the program shows the hash codes assigned by this method to the search key and the elements in the set:

Click here to view code image

```
Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

The hash codes of two objects, which are equal according to the `equals()` method of the class `UsableVNO`, are not equal according to the `hashCode()` method of the `Object` class. Therefore, the version number `(9.1.1)` in the set has a different hash code than the search key `(9.1.1)`. These objects hash to different buckets. The lookup for the search key is done in one bucket and does not find the element `(9.1.1)`, which is to be found in a completely different bucket. Just overriding the `equals()` method is not enough. The class `UsableVNO` violates the key tenet of the `hashCode()` contract: *Equal objects must produce equal hash codes*.

Example 14.5 *Implications of Not Overriding the* `Object.hashCode()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO2 {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(UsableVNO.class);

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                              // (
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                            // (

    // Create a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                     // (

    // Searching in a set:
    Set<UsableVNO> vnoSet = new HashSet<>(vnoList);                        // (
    out.println("Set:  " + vnoSet);
    out.printf("Search key %s contained in set:  %s%n", searchKey,
                vnoSet.contains(searchKey));                               // (
    out.println();
    // Search key and its hash code:
    out.printf("Search key %s has hash code: %d%n", searchKey,
               searchKey.hashCode());                                    // (

    // Hash values for elements:
    out.println("Hash codes for the elements:");
    for (UsableVNO element : versions) {                                 // (
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
  }
}
```

Output from the program:

Click here to view code image

```
class UsableVNO
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

General Contract of the `hashCode()` Method

The general contract of the `hashCode()` method stipulates the following:

- *Consistency during execution is a necessary prerequisite*: Multiple invocations of the `hashCode()` method on an object must consistently return the same hash code during the execution of an application, provided the object is not modified to affect the result returned by the `equals()` method. The hash code need not remain consistent across different executions of the application. This means that using a pseudorandom number generator to produce hash codes is not a valid strategy.
- *Object value equality implies hash code equality*: If two objects are equal according to the `equals()` method, then the `hashCode()` method must produce the same hash code for these objects. This tenet ties in with the general contract of the `equals()` method.
- *Object value inequality places no restrictions on the hash code*: If two objects are unequal according to the `equals()` method, then the `hashCode()` method need not produce distinct hash codes for these objects. However, it is

strongly recommended that the `hashCode()` method produce unequal hash codes for unequal objects.

Note that the hash code contract does not imply that objects with equal hash codes are equal. Not producing unequal hash codes for unequal objects can have an adverse effect on performance, as unequal objects with the same hash code will hash to the same bucket.

Heuristics for Implementing the `hashCode()` Method

In Example 14.6, the computation of the hash code in the `hashCode()` method of the `ReliableVNO` class embodies heuristics that can produce fairly reasonable hash functions. The hash code is computed according to the following formula:

Click here to view code image

$$\mathtt{hashValue = 11 * 31^3 + release * 31^2 + revision * 31^1 + patch}$$

Only the significant fields that have bearing on the `equals()` method are included. This ensures that objects that are equal according to the `equals()` method also have equal hash codes according to the `hashCode()` method.

Example 14.6 *Implementing the* `hashCode()` *Method*

Click here to view code image

```
import java.util.Objects;

// Overrides both equals() and hashCode().
public class ReliableVNO {

  private int release;
  private int revision;
  private int patch;

  public ReliableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                       // (1)
    return (this == obj)                                                // (2)
        || (obj instanceof ReliableVNO vno                              // (3)
            && this.patch    == vno.patch                          // (4)
            && this.revision == vno.revision
            && this.release  == vno.release);
  }

  @Override public int hashCode() {                                // (5)
    int hashValue = 11;
    hashValue = 31 * hashValue + release;
    hashValue = 31 * hashValue + revision;
    hashValue = 31 * hashValue + patch;
    return hashValue;

//  return Objects.hash(this.release, this.revision, this.patch); // (6)
  }
}
```

The basic idea is to compute an `int` hash code `sfVal` for each significant field `sf`, and include an assignment of the form shown at (1) in the computation below:

Click here to view code image

```
public int hashCode() {
  int sfVal;
  int hashValue = 11;
  ...
  sfVal = ...   // Compute hash code for each significant field sf. See below.
  hashValue = 31 * hashValue + sfVal;   // (1)
  ...
  return hashValue;
}
```

This setup ensures that the result from incorporating a field value is used to calculate the contribution from the next field value.

Calculating the hash code `sfVal` for a significant field `sf` depends on the type of the field:

- Field `sf` is a `boolean`:

  ```
  sfVal = sf ? 0 : 1;
  ```

- Field `sf` is a `byte`, `char`, `short`, or `int`:

  ```
  sfVal = (int)sf;
  ```

- Field `sf` is a `long`:

  Click here to view code image

  ```
  sfVal = (int) (sf ^ (sf >>> 32));
  ```

- Field `sf` is a `float`:

  ```
  sfVal = Float.floatToInt(sf);
  ```

- Field `sf` is a `double`:

  Click here to view code image

  ```
  long sfValTemp = Double.doubleToLong(sf);
  sfVal = (int) (sfValTemp ^ (sfValTemp >>> 32));
  ```

- Field `sf` is a reference that denotes an object. Typically, the `hashCode()` method is invoked recursively if the `equals()` method is invoked recursively:

  Click here to view code image

  ```
  sfVal = (sf == null ? 0 : sf.hashCode());
  ```

- Field `sf` is an array. The contribution from each element is calculated similarly to a field.

The order in which the fields are incorporated into the hash code computation will influence the hash code. Fields whose values are derived from other fields can be excluded. There is no point in feeding the hash function with redundant information, since this is unlikely to improve the value distribution. Fields that are not significant for the `equals()` method must be excluded; otherwise, the `hashCode()`

method might end up contradicting the `equals()` method. As with the `equals()` method, data from unreliable resources (e.g., network access) should not be used, and inclusion of transient fields should be avoided.

A legal or correct hash function does not necessarily mean it is appropriate or efficient. The classic example of a legal but inefficient hash function is

```
public int hashCode() {
  return 1949;
}
```

All objects using this method are assigned to the same bucket. The hash table is then no better than a list. For the sake of efficiency, a hash function should strive to produce unequal hash codes for unequal objects.

For numeric wrapper types, the `hashCode()` implementation returns an `int` representation of the primitive value, converting the primitive value to an `int`, if necessary. The `Boolean` objects for the `boolean` literals `true` and `false` have specific hash codes, which are returned by the `hashCode()` method.

The `hashCode()` method of the `String` class returns a hash code that is the value of a polynomial whose variable has the value 31; the coefficients are the characters in the string, and the degree is the string length minus 1. For example, the hash code of the string `"abc"` is computed as follows:

Click here to view code image

```
hashValue = 'a' * 31^2 + 'b' * 31^1 + 'c' * 31^0 = 97 * 31 * 31 + 98 * 31 + 99 = 9
```

For immutable objects, the hash code can be cached—that is, calculated once and returned whenever the `hashCode()` method is called.

However, the `hash()` method of the `Objects` class makes it very convenient to compute the hash code for an object with multiple fields. The procedure outlined above for calculating the hash code for a version number can be replaced by a single statement, as shown at (6) in Example 14.6. Each `int` field value is autoboxed in an `Integer` and passed in the call to the `Objects.hash()` method which returns a hash code based on the field values of the version number.

Click here to view code image

```
return Objects.hash(this.release, this.revision, this.patch); // (6)
```

The client in Example 14.7 demonstrates searching for objects of the class `ReliableVNO` in a set and in a map. The `ReliableVNO` objects in the array `versions` are used as elements in the set and as keys in the map.

Output from the program in Example 14.7 shows that the search key `(9.1.1)` and the element `(9.1.1)` in the set `vnoSet` have the same hash code. The search is successful. These objects hash to the same bucket. Therefore, the search for the element takes place in the right bucket. It finds the element `(9.1.1)` using the `equals()` method by successfully checking for equality between the search key `(9.1.1)` and the element `(9.1.1)` of the set.

Each entry in the map `versionStatistics` represents a version number (the *key*) and the number of downloads (the *value*) for that version number. Output from the program also shows that the key `(9.1.1)` of the entry `(9.1.1)=6000` in the map has the same hash code as the search key `(9.1.1)`. The search is successful.

However, we still cannot use objects of the class `ReliableVNO` in *sorted* sets and maps, as no criteria for comparing the version numbers have been defined.

Example 14.7 *Implications of Implementing the* `hashCode()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestReliableVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(ReliableVNO.class);
    // An array of version numbers.
    ReliableVNO[] versions =  new ReliableVNO[] {                           //
        new ReliableVNO( 3,49, 1), new ReliableVNO( 8,19,81),
        new ReliableVNO( 2,48,28), new ReliableVNO(10,23,78),
        new ReliableVNO( 9, 1, 1)};

    // Search key and its hash code:
    ReliableVNO searchKey = new ReliableVNO( 9, 1, 1);                      //
    out.printf("Search key %s has hash code: %d%n", searchKey,
        searchKey.hashCode());                                              //

    // Print hash values:
    out.println("Hash codes:");
    for (ReliableVNO element : versions) {                                  //
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
    out.println();

    // Searching in a set:
    Set<ReliableVNO> vnoSet = new HashSet<>(Arrays.asList(versions));       //
    out.println("Set: " + vnoSet);
    out.printf("Search key %s contained in set: %s%n%n", searchKey,
                vnoSet.contains(searchKey));                                //

    // Searching in a map:
    Map<ReliableVNO, Integer> versionStatistics = new HashMap<>();          //
    versionStatistics.put(versions[0], 2000);
    versionStatistics.put(versions[1], 3000);
    versionStatistics.put(versions[2], 4000);
    versionStatistics.put(versions[3], 5000);
    versionStatistics.put(versions[4], 6000);
    out.println("Map: " + versionStatistics);                               //
    out.printf("Search key %s contained in map: %s%n", searchKey,
               versionStatistics.containsKey(searchKey));                   //
  }
}
```

Output from the program:

Click here to view code image

```
class ReliableVNO
Search key (9.1.1) has hash code: 336382
Hash codes:
    (3.49.1): 332104

   (8.19.81): 336059
   (2.48.28): 331139
```

```
 (10.23.78): 338102
    (9.1.1): 336382

Set: [(10.23.78), (2.48.28), (9.1.1), (3.49.1), (8.19.81)]
Search key (9.1.1) contained in set: true

Map: {(10.23.78)=5000, (2.48.28)=4000, (9.1.1)=6000, (3.49.1)=2000,
     (8.19.81)=3000}
Search key (9.1.1) contained in map: true
```

14.4 Implementing the `java.lang.Comparable<E>` Interface

In order to sort the objects of a class, it should be possible to *compare* the objects. A *total ordering* allows objects to be compared. The criteria used to do the comparison depend on what is meaningful for the class. For example, comparison of `Integer` objects is based on the `int` value in an `Integer` object, whereas comparison of `String` objects is based on the Unicode value of the characters comprising the string in a `String` object. There can be more than one total ordering to compare the objects of a class. For example, another total ordering for `String` objects can be case insensitive—that is, treats uppercase characters as lowercase when comparing `String` objects.

The total ordering for objects of a class that is designated as the *default* ordering is called the *natural ordering*. A class defines the natural ordering for its object by implementing the `java.lang.Comparable<E>` generic interface. Other total orderings can be defined by providing a *comparator* that implements the `java.util.Comparator<E>` generic interface.

We will look at the two generic interfaces `Comparable<E>` and `Comparator<E>` for comparing objects in the rest of this chapter. The `Comparable<E>` interface is implemented by the objects of the class —in other words, the class provides the implementation for comparing its objects according to its natural ordering. The class usually leaves the implementation of a total ordering to its clients who can decide which total ordering is desirable.

The general contract for the `Comparable<E>` interface is defined by its only abstract method `compareTo()`, making this interface technically a functional interface. However, it is not annotated with the `@FunctionalInterface` annotation, and it is not intended to be implemented by lambda expressions.

```
int compareTo(E other)
```

> Returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, respectively, based on the natural ordering. It throws a `ClassCastException` if the reference value passed in the argument cannot be compared to the current object. It throws a `Null-PointerException` if the argument is `null`.

Many of the standard classes in the Java SE APIs, such as the primitive wrapper classes, enum types, `String`, `LocalDate`, `LocalDateTime`, `LocalTime`, and `File`, implement the `Comparable<E>` interface. Objects implementing this interface can be used as

- Elements in a sorted set
- Keys in a sorted map
- Elements in lists that are sorted using the `Collections.sort()` or `List.sort()` method
- Elements in arrays that are sorted using the overloaded `Arrays.sort()` methods

The natural ordering for `String` objects (and `Character` objects) is lexicographical ordering—that is, their comparison is based on the Unicode value of each corresponding character in the strings. Objects of the `String` and `Character` classes will be lexicographically maintained as elements in a sorted set, or as keys in a sorted map that uses their natural ordering.

The natural ordering for objects of a numerical wrapper class is in ascending order of the values of the corresponding numerical primitive type. As elements in a sorted set or as keys in a sorted map that uses their natural ordering, the objects will be maintained in ascending order.

According to the natural ordering for objects of the `Boolean` class, a `Boolean` object representing the value `false` is less than a `Boolean` object representing the value `true`.

An implementation of the `compareTo()` method for the objects of a class should meet the following criteria:

- For any two objects of the class, if the first object is *less than*, *equal to*, or *greater than* the second object, then the second object must be *greater than*, *equal to*, or *less than* the first object, respectively—that is, the comparison is *anti-*

symmetric.

- All three comparison relations (*less than*, *equal to*, *greater than*) embodied in the `compareTo()` method must be *transitive*. For example, for any objects `obj1`, `obj2`, and `obj3` of a class, if `obj1.compareTo(obj2) > 0` and `obj2.compareTo(obj3) > 0`, then `obj1.compareTo(obj3) > 0`.
- For any two objects of the class that compare as equal, the `compareTo()` method must return the same result if these two objects are compared with any other object—that is, the comparison is *congruent*.
- The `compareTo()` method is strongly recommended (but not required) to be *consistent with equals*—that is, `(obj1.compareTo(obj2) == 0) == (obj1.equals(obj2))` is `true`. This is recommended if the objects will be maintained in sorted sets or sorted maps.

The magnitude of non-zero values returned by the `compareTo()` method is immaterial; the sign indicates the result of the comparison. The general contract of the `compareTo()` method augments the general contract of the `equals()` method, providing a natural ordering of the compared objects. The equality test of the `compareTo()` method has the same provisions as that of the `equals()` method.

Implementing the `compareTo()` method is not much different from implementing the `equals()` method. In fact, given that the functionality of the `equals()` method is a subset of the functionality of the `compareTo()` method, the `equals()` implementation can call the `compareTo()` method. This guarantees that the two methods are always consistent with each other.

Click here to view code image

```
@Override public boolean equals(Object obj) {
  return (this == obj)
      || (obj instanceof Whatever other
          && this.compareTo(other) == 0);
}
```

Example 14.8 *Implementing the* `compareTo()` *Method of the* `Comparable<E>` *Interface*

Click here to view code image

```
import java.util.Comparator;
import java.util.Objects;

public final class VersionNumber implements Comparable<VersionNumber> {

  private final int release;
  private final int revision;
  private final int patch;

  public VersionNumber(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                      // (1)
    return (this == obj)                                             // (2)
        || (obj instanceof VersionNumber vno                         // (3)
            && this.patch    == vno.patch
            && this.revision == vno.revision
            && this.release  == vno.release);
  }

  @Override public int hashCode() {
    return Objects.hash(this.release, this.revision, this.patch);    // (4)
  }

  @Override public int compareTo(VersionNumber vno) {                // (5)
    // Compare the release numbers.                                     (6)
    if (this.release != vno.release)
      return Integer.compare(this.release, vno.release);

    // Release numbers are equal,                                       (7)
    // must compare revision numbers.
    if (this.revision != vno.revision)
      return Integer.compare(this.revision, vno.revision);

    // Release and revision numbers are equal,                          (8)
    // patch numbers determine the ordering.
    return Integer.compare(this.patch, vno.patch);
  }

  /*
  @Override public int compareTo(VersionNumber vno) {                // (9a)
    return Comparator.comparingInt(VersionNumber::getRelease)        // (10a)
                     .thenComparingInt(VersionNumber::getRevision)   // (11a)
                     .thenComparingInt(VersionNumber::getPatch)      // (12a)
                     .compare(this, vno);                            // (13a)
  }
  */
}
```

A `compareTo()` method is seldom implemented to interoperate with objects of other classes. For example, this is the case for primitive wrapper classes and the `String` class. The calls to the `compareTo()` method in the last three statements below all result in a compile-time error.

Click here to view code image

```
Integer iRef = 10;
Double dRef = 3.14;
```

```
String str = "ten";
StringBuilder sb = new StringBuilder("ten");
boolean b1 = iRef.compareTo(str) == 0 ;  // compareTo(Integer) not applicable
                                         // arguments (String).
boolean b2 = dRef.compareTo(iRef) > 0;   // compareTo(Integer) not applicable
                                         // arguments (Double).
boolean b3 = sb.compareTo(str) == 0; // compareTo(StringBuilder) not applicabl
                                     // arguments (String).
```

A straightforward implementation of the `compareTo()` method for version numbers is shown in Example 14.8. Note the specification of the `implements` clause in the class header. By parameterizing the `Comparable<E>` interface with the `VersionNumber` type, the class declaration explicitly excludes comparison with objects of other types. Only `VersionNumber`s can be compared.

Click here to view code image

```
public final class VersionNumber implements Comparable<VersionNumber> {
  ...
  @Override public int compareTo(VersionNumber vno) {                // (5)
  ...
  }
  ...
}
```

The signature of the `compareTo()` method is `compareTo(VersionNumber)`. In order to maintain backward compatibility with non-generic code, the compiler inserts the following *bridge method* with the signature `compareTo(Object)` into the class (§11.11, p. 615).

Click here to view code image

```
public int compareTo(Object obj) {   // NOT A GOOD IDEA TO RELY ON THIS METHOD
  return this.compareTo((VersionNumber) obj);
}
```

In an implementation of the `compareTo()` method, the fields are compared with the most significant field first and the least significant field last. In the case of the version numbers, the release numbers are compared first, followed by the revision numbers, with the patch numbers being compared last. Note that the next lesser significant fields are only compared if the comparison of the previous higher significant fields yielded equality. Inequality between corresponding significant fields short-circuits the computation. If all significant fields are equal, a zero will be returned. This approach is shown in the implementation of the `compareTo()` method at (5) through (8) in Example 14.8.

Comparison of integer values in fields can be optimized. In the code for comparing the release numbers at (5) in Example 14.8, we have relied on implicit use of the `Integer.compareTo()` method called by the `Integer.compare()` method, and autoboxing of the `int` field values:

Click here to view code image

```
if (this.release != vno.release)
  return Integer.compare(this.release, vno.release);
// Next field comparison
```

The code above can be replaced by the following code for doing the comparison, which relies on the difference between `int` values:

Click here to view code image

```
int releaseDiff = release - vno.release;
if (releaseDiff != 0)
  return releaseDiff;
// Next field comparison
```

However, this code can break if the difference is a value not in the range of the `int` type.

Significant fields with non-`boolean` primitive values are normally compared using the relational operators `<` and `>`. For comparing significant fields denoting constituent objects, the main options are to either invoke the `compareTo()` method on them, or use a comparator.

A more compact and elegant implementation of the `compareTo()` method for version numbers is shown at (9a) in Example 14.8 that exclusively uses methods of the `Comparator<E>` interface (p. 769). The `compareTo()` method implementation below is an equivalent version of the `compareTo()` method at (9a) in Example 14.8, where the method chaining has been split up and the method references replaced with equivalent lambda expressions.

Click here to view code image

```
@Override public int compareTo(VersionNumber vno) {                 // (9b)
  Comparator<VersionNumber> c1
      = Comparator.comparingInt(vn -> vn.getRelease());             // (10b)
  Comparator<VersionNumber> c2
      = c1.thenComparingInt(vn -> vn.getRevision());                // (11b)
  Comparator<VersionNumber> c3
      = c2.thenComparingInt(vn -> vn.getPatch());                   // (12b)
//return c3.compare(this, vno);                                     // (13b)
  return Objects.compare(this, vno, c3);                            // (13c)
}
```

The method implementations at (9a) and (9b) essentially use a *conditional comparator* that applies its constituent comparators conditionally in the order in which it is composed. The lambda expressions specified as arguments in the method calls above at (10b), (11b), and (12b) will extract the `int` value of a particular field in a version number when the expression is executed.

- The call to the `static` method `comparingInt()` at (10b) returns a `Comparator` that compares the version numbers by the `int` value of their `release` field.
- The call to the `default` method `thenComparingInt()` at (11b) on `Comparator c1` returns a composed `Comparator` that first compares version numbers using `Comparator c1`, and if the comparison result is `0`, compares them by the `revision` field.
- Analogous to (11b), the call to the `default` method `thenComparingInt()` at (12b) on `Comparator c2` returns a composed `Comparator` that first compares version numbers using `Comparator c2`, and if the comparison result is `0`, compares them by the `patch` field.
- It is the call to the functional method `compare()` at (13b) on the composed `Comparator c3` that does the actual comparison on the current version number (`this`) and the argument object (`vno`) according to the natural ordering of the `int` values of their fields, and where the order of the individual field comparisons is given by the composed `Comparator c3`.

  Equivalently, the `return` statement at (13b) can be written as (13c), where the convenience method `Objects.compare()` takes as arguments the current version number (`this`), the argument object (`vno`), and the composed `Comparator c3`. The `Objects.compare()` method calls the functional method `compare()` on the composed `Comparator c3`, passing it the current version number (`this`) and the argument object (`vno`).

What is different about this implementation of version numbers is that the class `VersionNumber` now overrides both the `equals()` and the `hashCode()` methods, and implements the `compareTo()` method of the parameterized `Comparable<VersionNumber>` interface. In addition, the `compareTo()` method is consistent with the `equals()` method. Following general class design principles, the class has been declared `final` so that it cannot be extended.

Example 14.9 *Implications of Implementing the* `compareTo()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestVersionNumber {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(VersionNumber.class);

    // Create an unsorted array of version numbers:
    VersionNumber[] versions =  new VersionNumber[] {                       //
        new VersionNumber( 3,49, 1), new VersionNumber( 8,19,81),
        new VersionNumber( 2,48,28), new VersionNumber(10,23,78),
        new VersionNumber( 9, 1, 1)};
    out.println("Unsorted array: " + Arrays.toString(versions));

    // Create an unsorted list:
    List<VersionNumber> vnoList = Arrays.asList(versions);                  //
    out.println("Unsorted list:  " + vnoList);

    // Create an unsorted map:
    Map<VersionNumber, Integer> versionStatistics = new HashMap<>();        //
    versionStatistics.put(versions[0], 2000);
    versionStatistics.put(versions[1], 3000);
    versionStatistics.put(versions[2], 4000);
    versionStatistics.put(versions[3], 5000);
    versionStatistics.put(versions[4], 6000);
    out.println("Unsorted map: " + versionStatistics);

    // Sorted set:
    Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                  //
    out.println("Sorted set: " + sortedSet);

    // Sorted map:
    Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics);/
    out.println("Sorted map: " + sortedMap);

    // Sorted list:
    Collections.sort(vnoList);                                              //
    out.println("Sorted list:    " + vnoList);

    // Searching in sorted list:
    VersionNumber searchKey = new VersionNumber( 9, 1, 1);                  //
    int resultIndex = Collections.binarySearch(vnoList, searchKey);        //
    out.printf("Binary search in sorted list found key %s at index: %d%n",
                searchKey, resultIndex);

    // Sorted array:
    Arrays.sort(versions);                                                  //
    out.println("Sorted array:   " + Arrays.toString(versions));

    // Searching in sorted array:
    int resultIndex2 = Arrays.binarySearch(versions, searchKey);           //
    out.printf("Binary search in sorted array found key %s at index: %d%n",
                searchKey, resultIndex2);
  }
}
```

Output from the program:

Click here to view code image

```
class VersionNumber
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted map: {(10.23.78)=5000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
               (2.48.28)=4000}

Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}

Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted list found key (9.1.1) at index: 3
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted array found key (9.1.1) at index: 3
```

Example 14.9 is a client that uses the class `VersionNumber` from Example 14.8. Unlike previous attempts, the following code from Example 14.9 demonstrates that `VersionNumber` objects can now be maintained in sorted sets and maps. A sorted set is created at (4) based on the unsorted list `vnoList`, and a sorted map is created at (5) based on the unsorted map `versionStatistics`.

Click here to view code image

```
Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                    // (
...
Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics); // (
```

The output from executing this code shows that the elements in the set and the keys of the map are sorted in the natural ordering for version numbers:

Click here to view code image

```
Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}
```

By default, the class `TreeSet<E>` relies on its elements to implement the `equals()` method and the `compareTo()` method. The output from the program in Example 14.9 shows that the `TreeSet<VersionNumber>` maintains its elements sorted in the natural ordering dictated by the `compareTo()` method. Analogously, the output from the program in Example 14.9 shows that the `TreeMap<VersionNumber, Integer>` maintains its entries sorted on the keys which are in the natural ordering dictated by the `compareTo()` method.

We can run generic operations on collections of version numbers. Utility methods provided by the `Collections` and `Arrays` classes in the `java.util` package are discussed in §15.11, p. 856, and §15.12, p. 864, respectively.

The following code sorts the elements in the list `vnoList` created at (2) in Example 14.9 according to their natural order:

Click here to view code image

```
Collections.sort(vnoList);                                                    // (6)
```

The output from executing this code shows that the elements in the list are indeed sorted in ascending order:

Click here to view code image

```
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

A binary search can be run on this sorted list to find the index of the version number `(9.1.1)` referenced by the reference `searchKey` at (7) in Example 14.9:

Click here to view code image

```
VersionNumber searchKey = new VersionNumber( 9, 1, 1);                        // (7)
int resultIndex = Collections.binarySearch(vnoList, searchKey);               // (8)
```

Natural ordering is assumed for the elements in the list. Executing the code prints the correct index of the search key in the sorted list:

Click here to view code image

```
Binary search in sorted list found key (9.1.1) at index: 3
```

Finally, the code in Example 14.9 sorts the elements in the array `versions` created at (1) according to their natural order:

Click here to view code image

```
Arrays.sort(versions);                                                       // (9)
```

The output from executing this code shows that the elements in the array are sorted as expected in ascending order:

Click here to view code image

```
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

We can run a binary search on this sorted list:

Click here to view code image

```
int resultIndex2 = Arrays.binarySearch(versions, searchKey);          // (10)
```

Again, natural ordering is assumed for the elements in the array. Executing the code prints the correct index of the search key in the sorted array:

Click here to view code image

```
Binary search in sorted array found key (9.1.1) at index: 3
```

14.5 Implementing the `java.util.Comparator<E>` Interface

The `java.util.Comparator<E>` interface is a *functional interface* and is designated as such with the `@FunctionalInterface` annotation in the Java SE API documentation— in other words, it is intended to be implemented by lambda expressions. Apart from its sole abstract method `compare()`, it defines a number of useful `static` and `default` methods which are listed at the end of this section. Several of these methods are illustrated throughout this chapter.

Precise control of ordering can be achieved by creating a customized comparator that imposes a specific total ordering on the elements. All comparators implement the `Comparator<E>` interface, providing implementation for its abstract method:

> ```
> int compare(E o1, E o2)
> ```
>
> The `compare()` method returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object, according to the total ordering —that is, its contract is equivalent to that of the `compareTo()` method of the `Comparable<E>` interface. Since this method tests for equality, it is strongly recommended that its implementation does not contradict the semantics of the `equals()` method for the objects.

An alternative ordering to the default natural ordering can be specified by passing a `Comparator` to the constructor when the sorted set or map is created. The `Collections` and `Arrays` classes provide utility methods for sorting and searching, which also take a `Comparator` (§15.11, p. 856, and §15.12, p. 864).

Example 14.10 demonstrates the use of different comparators for strings. The program creates several empty sorted sets, using the `TreeSet<E>` class, where the comparator is passed to the constructor ((1b), (1c), (4)). Elements from the `words` array are added to each sorted set by the `Collections.addAll()` method, as at (6). A text representation of each sorted set is then printed, as at (7). The output shows the sort order in which the elements are maintained in the set.

Example 14.10 *Natural Ordering and Total Orderings*

Click here to view code image

```
import java.util.*;

public class ComparatorUsage {
  public static void main(String[] args) {

    String[] words = {"court", "Stuart", "report", "Resort",             // (1)
                      "assort", "support", "transport", "distort"};

    // Choice of comparator.
    Set<String> strSet1 = new TreeSet<>();             // (1a) Natural ordering
    Set<String> strSet2 = new TreeSet<>(String.CASE_INSENSITIVE_ORDER); // (1b
    Set<String> strSet3 = new TreeSet<>(               // (1c) Rhyming ordering
      (String obj1, String obj2) -> {
        // Create reversed versions of the strings:                         (2)
        String reverseStr1 = new StringBuilder(obj1).reverse().toString();
        String reverseStr2 = new StringBuilder(obj2).reverse().toString();
        // Compare the reversed strings lexicographically.
        return reverseStr1.compareTo(reverseStr2);                       // (3)
      }
    );
    Set<String> strSet4 = new TreeSet<>(
        Comparator.comparingInt(String::length)        // (4) First length, the
                  .thenComparing(Comparator.naturalOrder())// (5) natural orde
    );

    // Add the elements from the words array to a set and print the set:
    Collections.addAll(strSet1, words);                                  // (6)
    System.out.println("Natural order:\n" + strSet1);                    // (7)
    Collections.addAll(strSet2, words);
    System.out.println("Case insensitive order:\n" + strSet2);
    Collections.addAll(strSet3, words);
    System.out.println("Rhyming order:\n" + strSet3);
    Collections.addAll(strSet4, words);
    System.out.println("Length, then natural order:\n" + strSet4);
  }
}
```

Output from the program:

Click here to view code image

```
Natural order:
[Resort, Stuart, assort, court, distort, report, support, transport]
Case insensitive order:
```

```
[assort, court, distort, report, Resort, Stuart, support, transport]
Rhyming order:
[Stuart, report, support, transport, Resort, assort, distort, court]
Length, then natural order:
[court, Resort, Stuart, assort, report, distort, support, transport]
```

The `String` class implements the `Comparable<E>` interface, providing an implementation of the `compareTo()` method. The `compareTo()` method defines the natural ordering for strings, which is lexicographical. The natural ordering is used to maintain the elements sorted lexicographically when the sorted set at (1a) is used. If we wish to maintain the strings in a different ordering, we need to provide a customized comparator.

The `String` class provides a static field (`CASE_INSENSITIVE_ORDER`) that denotes a comparator object with a `compare()` method that ignores the case when comparing strings lexicographically. This particular total ordering is used to maintain the elements sorted when the sorted set at (1b) is used. The comparator is passed as an argument to the set constructor. The output shows how the elements are maintained sorted in the set by this total ordering, which is a *case-insensitive ordering*.

We can create a string comparator that enforces *rhyming ordering* on the strings. In rhyming ordering, two strings are compared by examining their corresponding characters at each position in the two strings, starting with the characters in the *last* position. First the characters in the last position are compared, then those in the last but one position, and so on. For example, given the two strings `"report"` and `"court"`, the last two characters in both strings are the same. Continuing toward the start of the two strings, the character `'o'` in the first string is less than the character `'u'` in the second string. According to rhyming ordering, the string `"report"` is less than the string `"court"`.

Comparing two strings according to rhyming ordering is equivalent to reversing the strings and comparing the reversed strings lexicographically. If we reverse the two strings `"report"` and `"court"`, the reversed string `"troper"` is lexicographically less than the reversed string `"truoc"`.

A rhyming ordering comparator is implemented by the lambda expression at (1c) in Example 14.10. The lambda expression first creates reversed versions of the strings passed as arguments. A reversed version of a string is created using a string builder, which is first reversed and then converted back to a string, as shown at (2). The `compareTo()` method call at (3) compares the reversed strings, as the lexicographical ordering for the reversed strings is equivalent to the rhyming ordering for the original strings. This particular total ordering is used to maintain the elements sorted when the sorted set at (1c) is used. The lambda expression is passed as an argument to the set constructor, and is executed when the `compare()` method of the `Comparator<String>` interface is called by the sorted set implementation. The output shows how the elements are maintained sorted in the set by this total ordering, which is *rhyming ordering*.

Finally, a conditional comparator is composed at (4) and (5) that first compares the strings by their length, and if the lengths are equal, it employs the natural ordering for strings. The output shows that the strings are first sorted according to their lengths, and strings with equal lengths are sorted according to their natural ordering.

Example 14.11 illustrates using a comparator that orders version numbers (Example 14.8, p. 763) according to their reverse natural ordering. The method `Comparator.reversedOrder()` readily returns a comparator that imposes the reverse of the natural ordering.

A list of version numbers is initialized at (2). This list is sorted using the reverse natural ordering at (3). A binary search is done in this list at (4). We have used the same comparator for the search as we did for the sorting, in order to obtain predictable results. Searching this list with natural ordering at (5) does not find the key.

Example 14.11 *Using a Comparator for Version Numbers*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class UsingVersionNumberComparator {

  public static void main(String[] args) {
    VersionNumber[] versions = new VersionNumber[] {                         //
        new VersionNumber(3, 49, 1), new VersionNumber(8, 19, 81),
        new VersionNumber(2, 48, 28), new VersionNumber(10, 23, 78),
        new VersionNumber(9, 1, 1) };

    List<VersionNumber> vnList = new ArrayList<>();
    Collections.addAll(vnList, versions);                                    //
    out.println("List before sorting:\n  " + vnList);
    Collections.sort(vnList, Comparator.reverseOrder());                     //
    out.println("List after sorting according to " +
                "reverse natural ordering:\n  " + vnList);

    VersionNumber searchKey = new VersionNumber(9, 1, 1);
    int resultIndex = Collections.binarySearch(vnList, searchKey,
                                               Comparator.reverseOrder());  //
    out.printf("Binary search in list using reverse natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);

    resultIndex = Collections.binarySearch(vnList, searchKey);              //
    out.printf("Binary search in list using natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);
  }
}
```

Output from the program:

Click here to view code image

```
List before sorting:
  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
List after sorting according to reverse natural ordering:
  [(10.23.78), (9.1.1), (8.19.81), (3.49.1), (2.48.28)]
Binary search in list using reverse natural ordering found key (9.1.1) at inde
Binary search in list using natural ordering found key (9.1.1) at index: -6
```

The `Comparator<E>` interface also provides many useful `static` and `default` methods, including composing conditional comparators that can compare on multiple fields. Reference is given in the description below to where the methods are used.

Click here to view code image

```
default Comparator<T> reversed()
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

The first method returns a comparator that imposes the reverse ordering of this comparator, equivalent to `(a, b) -> this.compare(b, a)`.

The second method returns a comparator that compares `Comparable` objects in natural order, equivalent to `(a, b) -> a.compareTo(b)`.

The third method returns a comparator that imposes the reverse of the natural ordering on `Comparable` objects, equivalent to `(a, b) -> b.compareTo(a)`.

See Example 14.10, p. 770, and Example 14.11, p. 772.

Click here to view code image

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> cmp)
static <T> Comparator<T> nullsLast(Comparator<? super T> cmp)
```

Return a `null`-friendly comparator that considers `null` to be either less than non-`null` or greater than non-`null`, respectively. These are useful comparators for sorting or searching in collections and maps when `null`s are considered as actual values.

Click here to view code image

```
static <T, U> Comparator<T>
        comparing(Function<? super T,? extends U> func,
                  Comparator<? super U> cmp)
```

Returns a `Comparator<T>` that applies `func` to the two given elements and compares the results using the specified `Comparator cmp`. It is effectively equivalent to `(a, b) -> cmp.compare(func.apply(a), func.apply(b))`.

Click here to view code image

```
static <T, U extends Comparable<? super U>> Comparator<T>
        comparing(Function<? super T,? extends U> func)
```

Returns a `Comparator<T>` that applies `func` to the two given elements first, before comparing the results by natural ordering. It is effectively equivalent to `(a, b) -> func.apply(a).compareTo(func.apply(b))`.

Click here to view code image

```
static <T> Comparator<T>
        comparingPrimType(ToPrimTypeFunction<? super T> func)
```

Returns a `Comparator<T>` that applies `func` to the two given elements first, before comparing the primitive-value results.

PrimType is either an `Int`, `Long`, or `Double`.

See Example 14.8, p. 763, and Example 14.10, p. 770.

Click here to view code image

```
default Comparator<T>
        thenComparing(Comparator<? super T> cmp)
```

Returns a *conditional* comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, the elements are compared using the specified `Comparator cmp`. Effectively, this method first executes `this.compare(a, b)`, and then `cmp.compare(a, b)` if necessary.

See Example 14.10, p. 770.

Click here to view code image

```
default <U> Comparator<T>
        thenComparing(Function<? super T,? extends U> func,
                      Comparator<? super U> cmp)
```

Returns a conditional comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, it then applies `func` to the elements and the results are compared using the specified `Comparator cmp`. Effectively, this method first executes `this.compare(a, b)`, and then `cmp.compare (func.apply(a), func.apply(b))` if necessary.

Click here to view code image

```
default <U extends Comparable<? super U>> Comparator<T>
        thenComparing(Function<? super T,? extends U> func)
```

Returns a conditional comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, it then applies `func` to the elements and the results are compared by natural ordering. Effectively, this method first executes `this.compare(a, b)`, and then `func.apply(a).compareTo (func.apply(b))` if necessary.

Click here to view code image

```
default Comparator<T>
        thenComparingPrimType(ToPrimTypeFunction<? super T> func)
```

`PrimType` is either an `Int`, `Long`, or `Double`.

These primitive-type specialized methods return a conditional comparator that first determines, using `this Comparator`, if two given elements are equal. If they are equal, it then applies `func` to the two elements and the primitive-value results are compared.

See Example 14.8, p. 763, and Example 14.10, p. 770.

Review Questions

14.1 Which of the following statements are true about the `hashCode()` and `equals()` methods?

Select the two correct answers.

- **a.** Two objects that are different according to the `equals()` method must have different hash codes.
- **b.** Two objects that are equal according to the `equals()` method must have the same hash code.

c. Two objects that have the same hash code must be equal according to the `equals()` method.

d. Two objects that have different hash codes must be unequal according to the `equals()` method.

14.2 Given that the objects referenced by the parameters override the `equals()` and `hashCode()` methods appropriately, which return values are possible from the following method?

Click here to view code image

```
String func(Object x, Object y) {
  return (x == y) + " " + x.equals(y) + " " + (x.hashCode() == y.hashCod
}
```

Select the four correct answers.

a. `"false false false"`
b. `"false false true"`
c. `"false true false"`
d. `"false true true"`
e. `"true false false"`
f. `"true false true"`
g. `"true true false"`
h. `"true true true"`

14.3 Which code, when inserted at (1), in the `equalsImpl()` method will fulfill the contract of the `equals()` method?

Click here to view code image

```
public class Pair {
  private int a, b;
  public Pair(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public boolean equals(Object o) {
    return (this == o) || (o instanceof Pair) && equalsImpl((Pair) o);
  }

  private boolean equalsImpl(Pair o) {
    // (1) INSERT CODE HERE
  }
}
```

Select the three correct answers.

a. `return a == o.a || b == o.b;`
b. `return false;`
c. `return a >= o.a;`
d. `return a == o.a;`
e. `return a == o.a && b == o.b;`

14.4 Which code, when inserted at (1), will provide a correct implementation of the `hashCode()` method in the following program?

Click here to view code image

```
import java.util.*;
public class Measurement {
  private int count;
  private int accumulated;
  public Measurement() {}
  public void record(int v) {
    count++;
    accumulated += v;
  }
  public int average() {
    return accumulated/count;
  }
  public boolean equals(Object other) {
    if (this == other)
      return true;
    if (!(other instanceof Measurement))
      return false;
    Measurement o = (Measurement) other;
    if (count != 0 && o.count != 0)
      return average() == o.average();
    return count == o.count;
  }
  public int hashCode() {
    // (1) INSERT CODE HERE
  }
}
```

Select the two correct answers.

a. `return 31337;`

b. `return accumulated / count;`

c. Click here to view code image

```
return (count << 16) ^ accumulated;
```

d. `return ~accumulated;`

e. Click here to view code image

```
return count == 0 ? 0 : average();
```

14.5 Which statement is true about the following program?

Click here to view code image

```
import java.util.Comparator;
import java.util.Comparator;
class Person  implements Comparable<Person > {
  private String name;

  private int age;
  Person (String name, int age) { this.name = name; this.age = age; }

  public int compareTo(Person  p2) {
    Comparator<String> strCmp = Person.cmp();
    int status = strCmp.compare(this.name, p2.name);
    if (status == 0) {
      Comparator<Integer> intCmp = Person.cmp();
      status = intCmp.compare(this.age, p2.age);
    }
    return status;
  }
  public static <E extends Comparable<E>> Comparator<E> cmp() {
    return  (e1, e2) -> e2.compareTo(e1);
  }
}

public class Main {
  public static void main(String[] args) {
    Person  p1 = new Person ("Tom", 20);
    Person  p2 = new Person ("Dick", 30);
    Person  p3 = new Person ("Tom", 40);
    System.out.println((p1.compareTo(p2) < 0) + " " + (p1.compareTo(p3)
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile. When run, it will throw an exception.

c. The program will compile. When run, it will print `true false`.

d. The program will compile. When run, it will print `true true`.

e. The program will compile. When run, it will print `false false`.

f. The program will compile. When run, it will print `false true`.

14.6 Which method returns a `Comparator<E>` that is not equivalent to the other six?

Click here to view code image

```
import java.util.Comparator;
class CompDecls {

  public static <E extends Comparable<E>> Comparator<E> cmp1() {     //
    return new Comparator<E>() {
      public int compare(E e1, E e2) { return e2.compareTo(e1); }
    };
  }

  public static <E extends Comparable<E>> Comparator<E> cmp2() {     //
    return (e1, e2) -> e2.compareTo(e1);
  }

  public static <E extends Comparable<E>> Comparator<E> cmp3() {     //
    return ((Comparator<E>)(e1, e2) -> e1.compareTo(e2)).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp4() {     //
    return Comparable::compareTo;
  }

  public static <E extends Comparable<E>> Comparator<E> cmp5() {     //
    return ((Comparator<E>)Comparable::compareTo).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp6() {     //
    Comparator<E> cmp =  Comparable::compareTo;
    return cmp.reversed();
  }
  public static <E extends Comparable<E>> Comparator<E> cmp7() {      //
    return Comparator.reverseOrder();
  }
}
```

Select the one correct answer.

a. (1)
b. (2)
c. (3)
d. (4)
e. (5)
f. (6)
g. (7)

14.7 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test14RQ8 {
  public static void main(String[] args) {
    Integer[] values = {-42,15,-23,19,11,71};
    Arrays.sort(values, (v1, v2) -> v1.toString().compareTo(v2.toString(
    System.out.println(Arrays.toString(values));
  }
}
```

What is the result?
Select the one correct answer.

a. `[71, 19, 15, 11, -23, -42]`
b. `[-23, -42, 11, 15, 19, 71]`
c. `[-42, -23, 11, 15, 19, 71]`
d. `[71, 19, 15, 11, -42, -23]`
e. `[-42, 15, -23, 19, 11, 71]`
f. The program will throw an exception at runtime.
g. The program will fail to compile.

14.8 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Album {
  private static List<Album> albums = new ArrayList<>();
  private String title;
  private Album(String title) { this.title = title; }
  public String toString() { return title; }
  public static void addAlbum(String title) {
    albums.add(new Album(title));
  }
  public static void sortAlbums(Comparator<Album> c) {
    albums.sort(c);
  }
  public static void processAlbums(Consumer<Album> c) {
        albums.forEach(c);
      }
    }
```

and

Click here to view code image

```
public class Test14RQ11 {
  public static void main(String[] args) {
    Album.addAlbum("New Songs");
    Album.addAlbum("More Songs");
    Album.addAlbum("Greatest Hits");
    Album.addAlbum("Old Songs");
    Album.sortAlbums((a1, a2) -> a1.toString().length() - a2.toString().
    Album.processAlbums(a -> {
      System.out.print(a.toString() + " ");
    });
  }
}
```

What is the result?

Select the one correct answer.

a. Click here to view code image

```
New Songs Old Songs More Songs Greatest Hits
```

b. Click here to view code image

```
New Songs More Songs Greatest Hits Old Songs
```

c. Click here to view code image

```
Greatest Hits More Songs Old Songs New Songs
```

d. Click here to view code image

```
Old Songs Greatest Hits More Songs New Songs
```

e. The program will throw an exception at runtime.

f. The program will fail to compile.

14.9 Given the following code:

Click here to view code image

```
import java.util.Objects;
public class Album1 {
  private String title;
  public Album1(String title) { this.title = title; }
  public int hashCode() {
    int hash = 7;
    hash = 29 * hash + Objects.hashCode(this.title);
    return hash;
  }
  public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj == null) { return false; }
    if (getClass() != obj.getClass()) { return false; }
    final Album1 other = (Album1) obj;
    return Objects.equals(this.title, other.title);
  }
}
```

and

Click here to view code image

```
public class LP extends Album1 {
  public LP(String title) { super(title); }
}
```

and

Click here to view code image

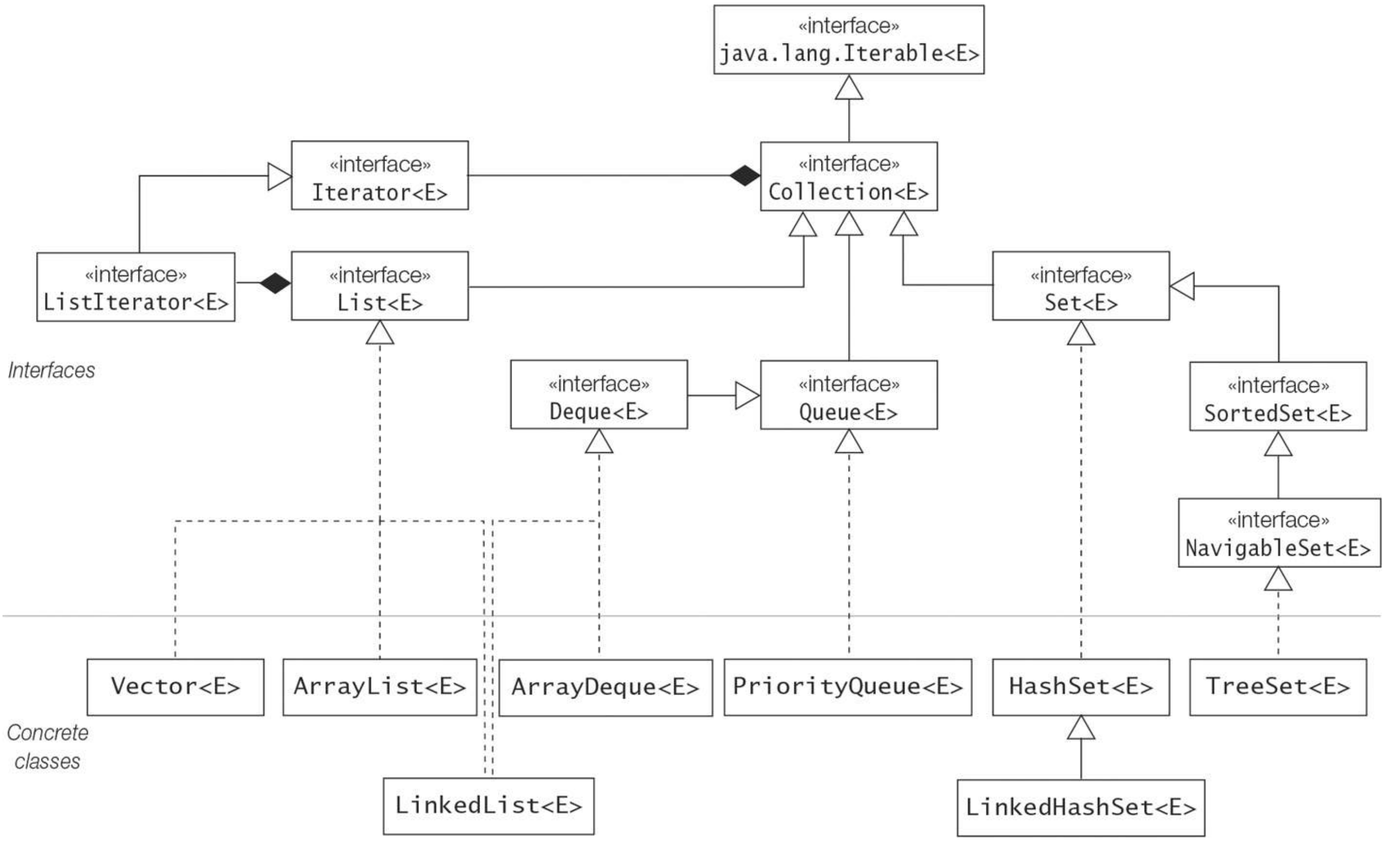
```
public class Test14RQ9 {
  public static void main(String[] args) {
    Album1 a1 = new Album1("Some Music");
    Album1 a2 = new LP("Some Music");
    if (a1.equals(a2)) {
      System.out.println("Same album");
    } else {
       System.out.println("These are different albums");
    }
  }
}
```

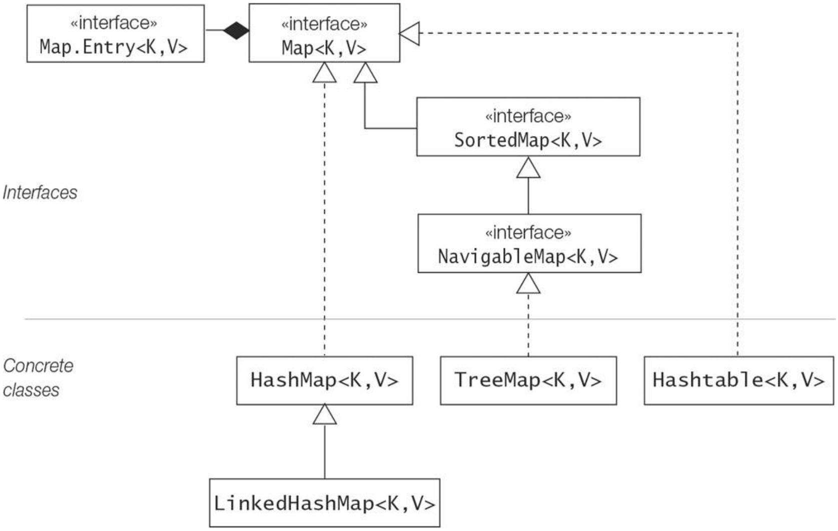
What is the result?
Select the one correct answer.

- **a.** `Same album`
- **b.** `These are different albums`
- **c.** The program will throw an exception at runtime.
- **d.** The program will fail to compile.

14.10 Which of the following statements are true about object ordering? Select the two correct answers.

- **a.** Class `A` can implement the `Comparator<A>` interface to compare its instances to other objects.
- **b.** Class `A` can implement the `Comparable<A>` interface to compare its instances to other objects.
- **c.** The `Comparator<A>` interface is implemented to establish a natural ordering for objects of class `A`.
- **d.** The `Comparable<A>` interface is implemented to establish a natural ordering for objects of class `A`.


«interface»
java.lang.Iterable<E>
«interface»
Iterator<E>
«interface»
Collection<E>
«interface»
ListIterator<E>
«interface»
List<E>
«interface»
Set<E>
Interfaces
«interface»
Deque<E>
«interface»
Queue<E>
«interface»
SortedSet<E>
«interface»
NavigableSet<E>
Vector<E>
ArrayList<E>
ArrayDeque<E>
PriorityQueue<E>
HashSet<E>
TreeSet<E>
Concrete
classes
LinkedList<E>
LinkedHashSet<E>


«interface»
Map.Entry<K,V>
«interface»
Map<K,V>
«interface»
SortedMap<K,V>
Interfaces
«interface»
NavigableMap<K,V>
Concrete classes
HashMap<K,V>
TreeMap<K,V>
Hashtable<K,V>
LinkedHashMap<K,V>

Collections: Part II

15

Chapter Topics

- An overview of the Java Collections Framework in the `java.util` package: core interfaces and their implementations
- Understanding the functionality provided by the `Collection<E>` interface, and its role in the Java Collections Framework
- How to create and use lists, and how their functionality is defined by the `List<E>` interface and implemented by the classes `Array-List<E>`, `Vector<E>`, and `LinkedList<E>`

- How to create and use sets, and how their functionality is defined by the `Set<E>` interface and implemented by the classes `HashSet<E>` and `LinkedHashSet<E>`
- How to create and use sorted and navigable sets, and how their functionality is defined by the `SortedSet<E>` and `NavigableSet<E>` interfaces, respectively, and implemented by the class `TreeSet<E>`
- How to create and use queues and deques, and how their functionality is defined by the `Queue<E>` and `Deque<E>` interfaces, and implemented by the classes `PriorityQueue<E>` and `ArrayDeque<E>`, respectively
- How to create and use maps, and how their functionality is defined by the `Map<K, V>` interface and implemented by the classes `HashMap<K, V>`, `LinkedHashMap<K, V>`, and `Hashtable<K, V>`
- How to create and use sorted and navigable maps, and how their functionality is defined by the `SortedMap<K, V>` and `NavigableMap<K, V>` interfaces, respectively, and implemented by the class `TreeMap<K, V>`
- Using the utility methods found in the `Collections` and `Arrays` classes, with emphasis on sorting and searching in lists and arrays

Java SE 17 Developer Exam O

[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements	*§15.1, p. 783 to §15.12, p. 865*

- *List, set, map, deque, and sorted collections and arrays are covered in this chapter.*
  - ○ *For arrays, see §3.9, p. 117.*
  - ○ *For `ArrayList`, see Chapter 12, p. 643.*
  - ○ *For comparing elements, see §14.4, p. 761, and §14.5, p. 769, respectively.*

Java SE 11 Developer Exam O

[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods	*§15.1, p. 783 to §15.9, p. 840*

- *List, set, map, and deque collections are covered in this chapter.*
  - ○ *For arrays, see §3.9, p. 117.*
  - ○ *For `ArrayList`, see Chapter 12, p. 643.*

[5.3] Sort collections and arrays using Comparator and Comparable interfaces *Sorting collections and arrays is covered in this chapter.* *For* `Comparable` *and* `Comparator` *interfaces, see §14.4, p. 761, and §14.5, p. 769, respectively.*	*§15.5, p. 810* *§15.10, p. 845* *§15.11, p. 858* *§15.12, p. 865*

The topics covered in this chapter can be divided into two main parts:

- In-depth coverage of collections, deques, and maps (§15.1 to §15.10)
- Sorting and searching in collections and arrays using the utility methods from the `Collections` and the `Arrays` classes (§15.11 and §15.12)

Some topics related to collections and maps are covered elsewhere in the book:

- The `ArrayList<E>` class is covered in Chapter 12, p. 643.
- Using streams with collections is covered in Chapter 16, p. 879.
- Thread-safe or concurrent collections and maps are covered in Chapter 22, p. 1365.
- Static methods from the utility class `Arrays` are used in many examples throughout the book.

15.1 The Java Collections Framework

A *collection* allows a group of objects to be treated as a single entity. Objects can be stored, retrieved, and manipulated as *elements* of a collection. Arrays are an example of one kind of collection.

Program design often requires the handling of collections of objects. The Java Collections Framework provides a set of standard utility classes for managing various kinds of collections. The core framework is provided in the `java.util` package and comprises three main parts:

- The core *interfaces* that allow collections to be manipulated independently of their implementation (see Figure 15.1 and Table 15.1). These *generic* interfaces define the common functionality exhibited by collections, and facilitate data exchange between collections.

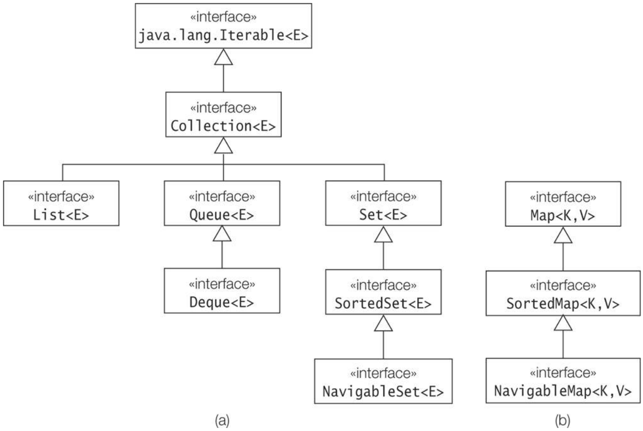

«interface»
java.lang.Iterable<E>
«interface»
Collection<E>
«interface»
List<E>
«interface»
Queue<E>
«interface»
Set<E>
«interface»
Map<K,V>
«interface»
Deque<E>
«interface»
SortedSet<E>
«interface»
SortedMap<K,V>
«interface»
NavigableSet<E>
«interface»
NavigableMap<K,V>
(a)
(b)

Figure 15.1 *The Core Interfaces*

Table 15.1 *Core Interfaces and Concrete Classes in the Collections Framework*

Interfaces	Concrete classes	Description
`Collection<E>`		A basic interface that defines the normal operations that allow a collection of objects to be maintained or handled as a single unit.
`List<E> extends Collection<E>`	`ArrayList<E>`, `Vector<E>`, `LinkedList<E>`	The `List<E>` interface extends the `Collection<E>` interface to maintain a sequence of elements that can contain duplicates.
`Set<E> extends Collection<E>`	`HashSet<E>`, `LinkedHashSet<E>`	The `Set<E>` interface extends the `Collection<E>` interface to represent its mathematical namesake: a *set* of unique elements.
`SortedSet<E> extends Set<E>`	`TreeSet<E>`	The `SortedSet<E>` interface extends the `Set<E>` interface to provide the required functionality for maintaining a set in which the elements are stored in some sort order.
`NavigableSet<E> extends SortedSet<E>`	`TreeSet<E>`	The `NavigableSet<E>` interface extends and replaces the `SortedSet<E>` interface to maintain a sorted set, and should be the preferred choice in new code.
`Queue<E> extends Collection<E>`	`PriorityQueue<E>`, `LinkedList<E>`	The `Queue<E>` interface extends the `Collection<E>` need to be processed according to some policy—that is, insertion at one end and removal at the other, usually as FIFO (first in, first out).
`Deque<E> extends Queue<E>`	`ArrayDeque<E>`, `LinkedList<E>`	The `Deque<E>` interface extends the `Queue<E>` interface to maintain a queue whose elements can be processed at both ends: *d*ouble-*e*nded *que*ue.
`Map<K,V>`	`HashMap<K,V>`, `Hashtable<K,V>`, `LinkedHashMap<K,V>`	A basic interface that defines operations for maintaining mappings of keys to values.
`SortedMap<K,V>`	`TreeMap<K,V>`	

The `SortedMap<K, V>` interface extends the `Map<K, V>` interface for maps that maintain their mappings sorted in key order.

`extends Map<K,V>`

The `NavigableMap<K, V>` interface extends and replaces the `SortedMap<K, V>` interface for sorted maps.

`NavigableMap<K,V>`
`extends`
`SortedMap<K,V>`

- A set of *implementations* (i.e., concrete classes, listed in Table 15.1) that are specific implementations of the core interfaces, providing data structures that a program can readily use.
- *Algorithms* that are an assortment of static *utility methods* found in the `Collections` and `Arrays` *classes* that can be used to perform various operations on collections and arrays, such as sorting and searching, or creating customized collections (§15.11, p. 856, and §15.12, p. 864).

Core Interfaces

The generic `Collection<E>` interface is a generalized interface for maintaining collections, and is the root of the interface inheritance hierarchy for collections shown in Figure 15.1a. These generic subinterfaces are summarized in Table 15.1. The `Collection<E>` interface extends the `Iterable<E>` interface that specifies an *iterator* to sequentially access the elements of an `Iterable` object (p. 791).

The elements in a `Set<E>` must be unique—that is, no two elements in the set can be equal. The order of elements in a `List<E>` is *positional*, and individual elements can be accessed randomly according to their position in the list.

Queues and deques, represented respectively by the `Queue<E>` and the `Deque<E>` interfaces, define collections whose elements can be processed according to various policies.

As can be seen in Figure 15.1b, the `Map<K, V>` interface does not extend the `Collection<E>` interface because conceptually, a map is not a collection. A map does not contain elements. It contains *mappings* (also called *entries*) from a set of *key* objects to a set of *value* objects. A key can, at most, be associated with one value—that is, it must be unique. As the name implies, the `SortedMap<K, V>` interface extends the `Map<K, V>` interface to maintain its mappings sorted in *key order*. It is superseded by the `NavigableMap<K, V>` interface which should be the preferred choice in new code.

Implementations

The `java.util` package provides implementations of a selection of well-known abstract data types, based on the core interfaces. Figures 15.2 and 15.3 show the inheritance relationship between the core interfaces and the corresponding implementations. None of the concrete implementations inherit directly from the `Collection<E>` interface. The abstract classes that provide the basis on which concrete classes are implemented are not shown in Figures 15.2 and 15.3.

Figure 15.2 *The Core Collection Interfaces and Their Implementations*

Figure 15.3 *The Core Map Interfaces and Their Implementations*

By convention, each of the collection implementation classes provides a constructor for creating a collection based on the elements of another `Collection` object passed as an argument. This allows the implementation of a collection to be changed by merely passing the collection to the constructor of the desired implementation. This interchangeability is also true between `Map` implementations. But collections and maps are not interchangeable. Note that a collection (or a map) only stores reference values of objects, and not the actual objects.

The collections framework is *interface-based*, meaning that collections are manipulated according to their interface types, rather than by the implementation types. By using these interfaces wherever collections of objects are used, various implementations can be used interchangeably.

All the concrete classes shown in Figures 15.2 and 15.3 implement the `Serializable` and the `Cloneable` interfaces; therefore, the objects of these classes can be serialized and also cloned.

A summary of collection and map implementations is given in Table 15.2. The contents of this table will be the focus as each core interface and its corresponding implementations are discussed in the subsequent sections.

Table 15.2 *Summary of Collection and Map Implementations*

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
[illegible]					
[illegible]	priority/deque order				
`Deque<E>`					
[illegible]					

[illegible] elements `hashCode()`	

[illegible] doubly linked list
`Set<E>`
`hashCode()`	

[illegible] elements (not `null`)
`Set<E>`
`hashCode()`,
`compareTo()`	

[illegible] (tree-like structure)
`Queue<E>`
`compareTo()`	

[illegible] `null`)	
[illegible] keys [illegible] `<K, V>` `hashCode()`	

[illegible] order of entries
`Map<K, V>`
`hashCode()`	

[illegible] keys (no `null` key and no `null` values)
`table<K, V>`
`hashCode()`	

[illegible] `null` key) `hashCode()`, `compareTo()`	

In Table 15.2, we see that only lists and queues allow *duplicates*—that is, the same value can be stored multiple times in the collection. The `TreeSet<E>` class and the queue classes `PriorityQueue<E>` and `ArrayDeque<E>` do *not* allow the `null` value to be stored in the collection. All maps require that the keys are *unique*—that is, no two keys can be equal, and only the classes `Hashtable<K, V>` and `TreeMap<K, V>` disallow the `null` value as a key. A `Hashtable<K, V>`, in addition, does not allow the `null` value to be associated with a key.

From Table 15.2, we also see that elements in a `LinkedHashSet<E>` are ordered, in a `TreeSet<E>` they are sorted, and in a `HashSet<E>` they have no order (i.e., they are *unordered*). *Sorting implies ordering* the elements in a collection according to some ranking criteria, usually based on the *values* of the elements. However, elements in an `ArrayList<E>` are maintained in the order they are inserted in the list, known as the *insertion order*. The elements in such a list are thus *ordered*, but they are *not* sorted, as it is not the values of the elements that determine their ranking in the list but their position in the list. Thus ordering does *not* necessarily imply sorting. In a `HashSet<E>`, the elements are unordered. No ranking of elements is implied in such a set. Whether a collection is sorted, ordered, or unordered also has implications when iterating over the collection (p. 791).

In order for sorting and searching to work properly, the collections and maps in Table 15.2 also require that their elements implement the appropriate methods for object comparison. Comparing objects is covered in Chapter 14, p. 741. Sorting and searching in collections is covered in §15.11, p. 856.

Methods defined for collections that specify functional interfaces as a parameter and require lambda expressions as arguments are covered in Chapter 13, p. 673.

Using streams with collections is covered in Chapter 16, p. 879.

The collection and map implementations discussed in this chapter, except for `Vector<E>` and `Hashtable<K, V>`, are not *thread-safe*—that is, their integrity can be jeopardized by concurrent access. Concurrent collections and maps are covered in Chapter 22, p. 1365.

The Java Collections Framework provides a plethora of collections and maps for use in single-threaded and concurrent applications, which should eliminate the need to implement one from scratch.

15.2 Collections

The `Collection<E>` interface specifies the contract that all collections should implement. Some of the operations in the interface are *optional*, meaning that a collection may choose to provide a stub implementation of such an operation that throws an `UnsupportedOperationException` when invoked—for example, when a collection is unmodifiable. The implementations of collections from the `java.util` package support all the optional operations in the `Collection<E>` interface (see Figure 15.2 and Table 15.2).

Many of the methods return a `boolean` value to indicate whether the collection was modified as a result of the operation.

Basic Operations

The basic operations are used to query a collection about its contents and allow elements to be added to and removed from a collection. Many examples in this chapter make use of these operations.

Click here to view code image

```
int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(E element)                   Optional
boolean remove(Object element)           Optional
```

The `size()` method returns the number of elements in the collection, and the `isEmpty()` method determines whether there are any elements in the collection.

The `contains()` method checks for membership of the argument object in the collection, using object value equality.

The `add()` and `remove()` methods return `true` if the collection was modified as a result of the operation.

By returning the value `false`, the `add()` method indicates that the collection excludes duplicates, and that the collection already contains an object equal to the argument object.

Note that we can only add an object of a specific type (`E`). However, a collection allows us to determine whether it has an element equal to an arbitrary object, or remove an element that is equal to an arbitrary object.

Bulk Operations

These operations are performed on a collection as a single entity. See §15.4, p. 804, for an example.

Click here to view code image

```
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c)        Optional
boolean removeAll(Collection<?> c)               Optional
boolean retainAll(Collection<?> c)               Optional
void    clear()                                  Optional
```

These bulk operations can be used to perform the equivalent of set logic on *arbitrary collections* (i.e., also lists and not just sets). The `containsAll()` method returns `true` if all elements of the specified collection are also contained in the current collection.

The `addAll()`, `removeAll()`, and `retainAll()` methods are *destructive* in the sense that the collection on which they are invoked can be modified. The operations performed by these methods are visualized by Venn diagrams in Figure 15.4.

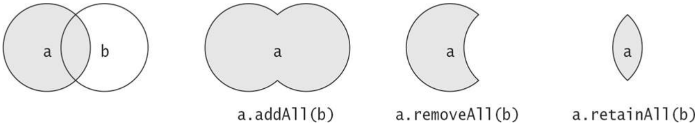


Figure 15.4 *Bulk Operations on Collections*

The `addAll()` method requires that the element type of the other collection is the same as, or a subtype of, the element type of the current collection. The `removeAll()` and `retainAll()` operations can be performed with collections of any type.

The `clear()` method removes all elements in the collection so that the collection is empty.

Iterating Over a Collection

A common operation on a collection is to iterate over all elements in a collection in order to perform a particular operation on each individual element. We explore several ways in which this operation can be implemented:

- Using the `hasNext()` and `next()` methods of the `Iterator<E>` interface (p. 793) to access the elements in the underlying collection
- Using the `forEachRemaining()` default method of the `Iterator<E>` interface to access the elements in the underlying collection
- Using the enhanced `for(:)` loop to iterate over an object that implements the `Iterable<E>` interface (see below)
- Using the `forEach()` default method of the `Iterable<E>` interface
- Using a sequential or parallel stream to iterate over a collection (p. 798)

The `Iterable<E>` interface defines the methods shown below. The enhanced `for(:)` loop can be used to iterate over objects that implement this interface. The `Collection<E>` interface extends the `Iterable<E>` interface, making all collections iterable.

Click here to view code image

```
Iterator<E> iterator()                                From Iterable<E> interface.
```

Returns an iterator that can be used to iterate over the elements in this collection. Any guarantee regarding the order in which the elements are returned is provided by the individual collection.

> Click here to view code image
>
> `default void forEach(Consumer<? super T> action)` From `Iterable<E>` interface.
>
> With this method, the specified `action` is performed on each of the remaining elements of an `Iterable`, unless an exception is thrown, which is then relayed to the caller. The functional interface `Consumer<T>` is covered in §13.7, p. 709.

Example 15.1 illustrates various ways to iterate over a collection that is created at (1). The example uses an `ArrayList<E>` for a collection of names that are `String`s. First an empty `ArrayList` is created and names are added to the collection. The operation performed on the collection prints each name in uppercase.

Example 15.1 *Iterating Over a Collection*

Click here to view code image

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

public class IterationOverCollections {
  public static void main(String[] args) {

    // Create a collection of names.                                   (1)
    Collection<String> collectionOfNames = new ArrayList<>();
    Collections.addAll(collectionOfNames, "Alice", "Bob", "Tom","Dick","Harrie
    System.out.println(collectionOfNames);

    // Using an explicit iterator.
    Iterator<String> iterator = collectionOfNames.iterator();        // (2)
    while (iterator.hasNext()) {                                      // (3)
      String name = iterator.next();                                  // (4)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();

    // Using the forEachRemaining() method of the Iterator interface.
    iterator = collectionOfNames.iterator();                          // (5)
    iterator.forEachRemaining(name ->                                 // (6)
        System.out.print(name.toUpperCase() + " "));
    System.out.println();

    // Using the for(:) loop on an Iterable.
    for (String name : collectionOfNames) {                           // (7)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();

    // Using the forEach() method of the Collection interface.
    collectionOfNames.forEach(name ->                                 // (8)
        System.out.print(name.toUpperCase() + " "));
    System.out.println();
    // Filtering using an explicit iterator.
    iterator = collectionOfNames.iterator();                         // (9)
    while (iterator.hasNext()) {                                     // (10)
      String name = iterator.next();                                 // (11)
      if (name.length() == 3)
        iterator.remove();                                           // (12)
    }
    System.out.println(collectionOfNames);

    // Filtering using the removeIf() method of the Collection interface.
    collectionOfNames.removeIf(name -> name.startsWith("A"));        // (13)
        System.out.println(collectionOfNames);

    // Using the stream() method of the Collection interface.
    collectionOfNames.stream()                                       // (14)
                     .forEach(name -> System.out.print(name.toUpperCase() + "
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
[Alice, Bob, Tom, Dick, Harriet]
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
[Alice, Dick, Harriet]
[Dick, Harriet]
DICK HARRIET
```

Using an Explicit Iterator on a Collection

A collection provides an iterator which allows sequential access to the elements of a collection. An iterator can be obtained by calling the `iterator()` method of the `Collection<E>` interface. An iterator is defined by the generic interface `java.util.Iterator<E>` that has the following methods:

> Click here to view code image
>
> `boolean hasNext()` From `Iterator<E>` interface.
>
> Returns `true` if the underlying collection still has elements left to return. A future call to the `next()` method will return the next element from the collection.
>
> Click here to view code image
>
> `E next()` From `Iterator<E>` interface.
>
> Moves the iterator to the next element in the underlying collection, and returns the current element. If there are no

more elements left to return, it throws a `NoSuchElementException`.

Click here to view code image

`void remove()`	Optional	From `Iterator<E>` interface.

Removes the element that was returned by the last call to the `next()` method from the underlying collection. Invoking this method results in an `Illegal-StateException` if the `next()` method has not yet been called, or when the `remove()` method has already been called after the last call to the `next()` method. This method is optional for an iterator—that is, it throws an `Unsupported-OperationException` if the remove operation is not supported.

Click here to view code image

```
default void
    forEachRemaining(Consumer<? super E> action)
```
From `Iterator<E>` interface.

The specified `action` is performed on each of the remaining elements in the collection associated with an iterator, unless an exception is thrown during its execution. The functional interface `Consumer<T>` is covered in §13.7, p. 709.

After obtaining the iterator for a collection, the methods provided by the `Iterator<E>` interface can be used systematically to iterate over the elements of the underlying collection one at a time.

In Example 15.1, an explicit iterator is obtained at (2) and used in the loop at (3) to iterate over all elements in the underlying collection. A call to the `hasNext()` method in the condition of the loop ensures that there is still an element to retrieve from the collection. At (4) the current element is retrieved by the iterator from the collection by calling the `next()` method. No casts are necessary at (4), as the compiler guarantees that the iterator will return a `String` object from the underlying collection.

Click here to view code image

```
Iterator<String> iterator = collectionOfNames.iterator();        // (2)
while (iterator.hasNext()) {                                      // (3)
  String name = iterator.next();                                  // (4)
  System.out.print(name.toUpperCase() + " ");
}
```

Note that the methods are invoked on the iterator, not the underlying collection. In Example 15.1, the `hasNext()` method is called *before* the `next()` method to ensure that there is still an element remaining to be processed; otherwise, a `java.util.NoSuchElementException` can be raised at runtime. Using an explicit iterator puts this responsibility on the user code.

In Example 15.1, we used an iterator in a `while` loop at (3) for iterating over the collection. It is quite common to use an iterator in a `for(;;)` loop for this purpose, where the iterator is obtained in the initialization part, and the increment part is left empty:

Click here to view code image

```
for (Iterator<String> iterator = collectionOfNames.iterator();
     iterator.hasNext(); /* Empty increment. */) {
  String name = iterator.next();
  System.out.print(name.toUpperCase() + " ");
}
```

The majority of the iterators provided in the `java.util` package are said to be *fail-fast*. When an iterator has already been obtained, structurally modifying the underlying collection by other means will invalidate the iterator. Subsequent use of this iterator will throw a `ConcurrentModificationException`, as the iterator checks to see if the collection has been structurally modified every time it accesses the collection. The `remove()` method of an iterator is the only recommended way to delete elements from the underlying collection during iteration with an iterator (p. 796).

The order in which the iterator will return the elements from an underlying collection depends on the iteration order supported by the collection. For example, an iterator for a list will iterate over the elements in the sequential order they have in the list, whereas the iteration order for the elements in an ordinary set is not predetermined. An iterator for a sorted collection will make the elements available in a given sort order. Iteration order will be discussed together with the individual concrete collection classes.

Using the `forEachRemaining()` *Method*

A convenient way to perform a task on each element of a collection is to use the `forEachRemaining()` method of the `Iterator<E>` interface. This method requires a consumer as an argument (§13.7, p. 709). The lambda expression defining the consumer is executed for each *remaining* element of the collection. In Example 15.1, a new iterator is obtained at (5) as the previous one was exhausted, before calling the method at (6). The lambda expression passed as an argument prints the name in uppercase (postfixed with a space for readability in the output). Note that the method is invoked on the iterator, and not directly on the collection.

Click here to view code image

```
iterator = collectionOfNames.iterator();                                // (5)
iterator.forEachRemaining(name ->                                       // (6)
    System.out.print(name.toUpperCase() + " "));
```

In Example 15.1, the `toString()` method of the collection class is used implicitly in print statements to generate a text representation for the collection. The collection classes override the `Object.toString()` method to provide a text representation of their contents. The standard text representation generated by the `toString()` method for a collection is

`[`*element*$_1$, *element*$_2$, ..., *element*$_n$`]`

where each *element*$_i$, where `1 <= i <= n`, is the text representation generated by the `toString()` method of the individual elements in the collection.

Using the `for(:)` *Loop to Iterate Over a Collection*

A `for(:)` loop can be used to iterate over an array or a data structure that implements the `java.lang.Iterable<E>` interface. This interface requires the implementation of the `iterator()` method to obtain an iterator that is used behind the scenes to iterate over elements of the data structure in a `for(:)` loop.

In the `for(:)` loop construct

`for` (*type variable* `:` *expression*`)` *statement*

the value of *expression* can be a reference value that refers to a collection that implements the `Iterable<E>` interface. In Figure 15.2 we see that the `Collection<E>` interface extends the `Iterable<E>` interface, and therefore, all collections that implement the `Collection<E>` interface can be iterated over using the `for(:)` loop. A collection that implements the `Collection<E>` interface and thereby the `Iterable<E>` interface has the element type `E`. This element type `E` must be assignable to the *type* of the *variable* in the `for(:)` loop. The *variable* is assigned the reference value of a new element in the collection each time the body of the loop is executed.

The semantics of the `for(:)` loop also apply when iterating over a collection. In particular, any structural change to the collection (adding or removing elements) in the `for(:)` loop will result in a `ConcurrentModificationException`—in other words, the underlying collection is structurally immutable.

Example 15.1 illustrates using a `for(:)` loop to iterate over a collection. The collection of names is iterated over in the `for(:)` loop at (7), printing each name in uppercase.

Click here to view code image

```
for (String name : collectionOfNames) {                                // (7)
  System.out.print(name.toUpperCase() + " ");
}
```

Behind the scenes, however, an appropriate iterator is used to iterate over the collection, but the `for(:)` loop simplifies iterating over a collection in the source code.

Note that if the collection is ordered or sorted, the iterator will iterate over the collection in the ordering used to maintain the elements in the collection. For example, in the case of an `ArrayList`, the iterator will yield the elements in the same order as the insertion order. In the case of a `TreeSet<E>`, the iterator will yield the elements in the sort order used to maintain the elements in the set. If the collection is unordered, the order in which the iterator will yield the elements is not predictable. Thus we cannot be sure in which order the elements of a `HashSet` will be iterated by the `for(:)` loop.

Using the `forEach()` *Method to Iterate Over a Collection*

The `default` method `forEach()` of the `Iterable<E>` interface (which the `Collection<E>` interface implements) allows us to do away with an explicit iterator in order to perform an operation on each element of the collection. The method requires a consumer as the argument (§13.7, p. 709). The lambda expression defining the consumer is executed successively for each element of the collection.

In Example 15.1, the lambda expression passed as the argument at (8) prints each name in uppercase. Note that the method is invoked directly on the collection.

Click here to view code image

```
collectionOfNames.forEach(name ->
    System.out.print(name.toUpperCase() + " "));                       // (8)
```

Filtering

We illustrate here two ways of filtering a collection—in this case, removing elements from the collection that satisfy certain criteria.

Using the `remove()` Method of an Iterator to Filter a Collection

Example 15.1 shows how an explicit iterator can be used in a loop to *remove* elements from a collection that fulfills certain criteria. As we have seen earlier, an explicit iterator is obtained at (9) and used in the loop at (10). The order of calls to the `hasNext()` and `next()` methods on the iterator at (10) and (11), respectively, ensures that each element in the underlying collection is retrieved in the loop. The call to the `remove()` method in the loop at (12) removes the current element from the underlying collection, if the criteria in the `if` statement is satisfied.

Click here to view code image

```
// [Alice, Bob, Tom, Dick, Harriet]
iterator = collectionOfNames.iterator();                          // (9)
while (iterator.hasNext()) {                                      // (10)
  String name = iterator.next();                                  // (11)
  if (name.length() == 3)
    iterator.remove();                                            // (12)
}
// [Alice, Dick, Harriet]
```

Best practices dictate that the three methods of the iterator should be used *in lockstep* inside a loop, as shown in Example 15.1. In particular, the `next()` method *must* be called *before* the `remove()` method for each element in the collection; otherwise, a `java.lang.IllegalStateException` or an `UnsupportedOperationException` is raised at runtime, depending on whether the collection provides this optional operation or not. As noted earlier, using an explicit iterator places the responsibility of bookkeeping on the user code.

Using the `removeIf()` Method to Filter a Collection

The following `default` method of the `Collection<E>` interface makes the task of filtering a collection much easier, as no explicit iteration is necessary:

> Click here to view code image
>
> ```
> default boolean removeIf(Predicate<? super E> filter)
> ```

> All elements that satisfy the predicate defined by the specified `filter` are removed from this collection. The method returns `true` or `false` depending on whether the collection was modified or not. Predicates are covered in §13.6, p. 703.

In Example 15.1, the call to the `removeIf()` method at (13) takes a `Predicate` to test whether a name starts with the letter `A`, resulting in those names satisfying the predicate to be removed from the collection.

Click here to view code image

```
// [Alice, Dick, Harriet]
collectionOfNames.removeIf(name -> name.startsWith("A"));         // (13)
// [Dick, Harriet]
```

Ample examples of filtering can also be found in §13.3, p. 691, and §16.5, p. 910.

Streams

The two methods of the `Collections` interface shown below allow a collection to be used as the *source of a stream*. A stream makes the elements of a collection available as a *sequence* of elements on which *sequential* and *parallel aggregate operations* can be performed.

> Click here to view code image
>
> ```
> default Stream<E> stream()
> default Stream<E> parallelStream()
> ```
>
> Return a sequential stream or a possibly parallel stream with this collection as its source, respectively.

Example 15.1 illustrates at (14) how a stream can be used to iterate over the collection of names in order to print them in uppercase.

Click here to view code image

```
collectionOfNames.stream()                                        // (14)
                 .forEach(name -> System.out.print(name.toUpperCase() + " "));
```

Details are revealed in §16.4, p. 897, which is devoted entirely to streams, providing many examples of using streams on collections.

Array Operations

The following operations convert collections to arrays. See also array operations on array lists (§12.6, p. 658).

> ```
> Object[] toArray()
> <T> T[] toArray(T[] a)
> ```
>
> The first `toArray()` method returns an array of type `Object` filled with all the elements of the collection. The second method is a generic method that stores the elements of a collection in an array of type `T`. The default method uses the `generator` function to allocate the returned array.
>
> If the given array is big enough, the elements are stored in this array. If there is room to spare in the array; that is, the length of the array is greater than the number of elements in the collection—the spare room is filled with `null` values before the array is returned. If the array is too small, a new array of type `T` and appropriate size is created. If `T` is not a supertype of the runtime type of every element in the collection, an `ArrayStoreException` is thrown.
>
> Click here to view code image
>
> ```
> default <T> T[] toArray(IntFunction<T[]> generator)
> ```
>
> Allows creation of an array of a particular runtime type given by the parameterization of the type parameter `T[]`, using the specified `generator` function (§13.8, p. 717) to allocate the array of the desired type and the specified length. The `default` implementation calls the `generator` function with 0 and then passes the resulting array of length 0 to the `toArray(T[])` generic method.

Example 15.2 illustrates converting a set to an array. At (1), the call to the non-generic `toArray()` method returns an array of type `Object`. Since an array of type `Object` is not a subtype of an array of type `String`, the compiler reports an error at (2).

At (3), the call to the generic `toArray()` method returns an array of size 3 and type `Object[]`, but element type `String`, when the method was passed a zero-length array of type `Object`. In other words, the method created a suitable-size array of type `Object`, since the array passed in the argument was too small. This array was filled with the elements of the set, which are strings. Although the array returned is of type `Object`, the objects stored in it are of type `String`. The output from the program confirms these observations.

At (4), the call to the generic `toArray()` method returns an array of size 3 and type `String[]`, having element type `String`, when the method was passed a zero-length array of type `String`. Now the method creates a new suitable-size array of type `String` and fills it with the elements of the set, which are strings. The output from the program shows that the array passed in the argument is not the same as the array returned by the method call.

At (5), the call to the generic `toArray()` method returns the same array it was passed in the argument, since it is of appropriate size and type. In other words, the array passed in the argument is filled with the elements of the list, and returned. This is corroborated by the output from the program.

At (6), the actual type parameter is `Integer`. The generic `toArray()` method throws an `ArrayStoreException` because `String` objects in the set denoted by `strSet` cannot be stored in an array of type `Integer`.

Lastly, Example 15.2 illustrates converting a set to an array using the default `toArray(IntFunction<T[]>)` generic method, which is a convenience method that actually leverages the `toArray(T[])` generic method.

Click here to view code image

```
IntFunction<String[]> createStrArray = nn -> new String[nn];          // (7)
String[] strArray5 = strSet.toArray(createStrArray);                  // (8a)
String[] strArray6 = strSet.toArray(String[]::new);                   // (8b)
String[] strArray7 = strSet.toArray(createStrArray.apply(0));         // (8c)
```

The lambda expression at (7) creates an array of length `nn` and of type `String[]`. This lambda expression is passed to the `toArray(IntFunction<String[]>)` method at (8a). Equivalently, we can pass a method reference to the method, as shown at (8b). Of course, the lambda expression at (7) only creates an array when the `apply()` method of the `IntFunction<T[]>` interface is called with a value for the length of the array. The default behavior of the `toArray(IntFunction<T[]>)` generic method is illustrated at (8c): The `apply()` method of the `IntFunction<String[]>` interface is explicitly called with the value `0` to create a zero-length `String` array which is then passed as a parameter to the `toArray(String[])` method.

Example 15.2 *Converting Collections to Arrays*

Click here to view code image

```
import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;
import java.util.function.IntFunction;

public class CollectionToArray {
  public static void main(String[] args) {

    Collection<String> strSet = new HashSet<>();
    strSet.add("2021"); strSet.add("2022"); strSet.add("2023");
    int n = strSet.size();

    Object[] objects = strSet.toArray();                // (1)
//  String[] string = strSet.toArray();                 // (2) Compile-time er

    Object[] objArray = strSet.toArray(new Object[0]);                     //
    System.out.println("Array size: " + objArray.length);
    System.out.println("Array type: " + objArray.getClass().getName());
    System.out.println("Actual element type: " +
                       objArray[0].getClass().getName());

    String[] strArray1 = new String[0];
    String[] strArray2 = strSet.toArray(strArray1);                        //
    System.out.println("strArray1 == strArray2: " + (strArray1 == strArray2));

    String[] strArray3 = new String[n];
    String[] strArray4 = strSet.toArray(strArray3);                        //
    System.out.println("strArray3 == strArray4: " + (strArray3 == strArray4));

//  Integer[] intArray = strSet.toArray(new Integer[n]);     // (6) Runtime er

    IntFunction<String[]> createStrArray = nn -> new String[nn];           //
    String[] strArray5 = strSet.toArray(createStrArray);                   //
    String[] strArray6 = strSet.toArray(String[]::new);                    //
    String[] strArray7 = strSet.toArray(createStrArray.apply(0));          //
    System.out.println("strArray5: " + Arrays.toString(strArray5));
    System.out.println("strArray6: " + Arrays.toString(strArray6));
    System.out.println("strArray7: " + Arrays.toString(strArray7));
  }
}
```

Output from the program:

Click here to view code image

```
Array size: 3
Array type: [Ljava.lang.Object;
Actual element type: java.lang.String
strArray1 == strArray2: false
strArray3 == strArray4: true
strArray5: [2023, 2022, 2021]
strArray6: [2023, 2022, 2021]
strArray7: [2023, 2022, 2021]
```

15.3 Lists

Lists are collections that maintain their elements *in order* and can contain duplicates. The elements in a list are *ordered*. Each element, therefore, has a position in the list. A zero-based index can be used to access the element at the position designated by the index value. The position of an element can change as elements are inserted or deleted from the list—that is, as the list is changed structurally.

The `List<E>` Interface

In addition to the operations inherited from the `Collection<E>` interface, the `List<E>` interface also defines operations that work specifically on lists: position-based access of the list elements, searching in a list, operations on parts of a list (called *open range-view* operations), and applying an operator to each element. These list operations are covered in Chapter 12, p. 643.

Sorting and searching in lists is covered in §15.11, p. 856.

The `List<E>` interface also provides the overloaded static method `of()` to create *unmodifiable* lists (§12.2, p. 649).

Additionally, the `List<E>` interface provides two customized list iterators:

> Click here to view code image
>
> ```
> ListIterator<E> listIterator()
> ListIterator<E> listIterator(int index)
> ```
>
> The iterator from the first method iterates over the elements consecutively, starting with the first element of the list, whereas the iterator from the second method starts iterating over the list from the element designated by the specified index.

The declaration of the `ListIterator<E>` interface is shown below:

Click here to view code image

```
interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  boolean hasPrevious();
  E next();                // Element after the cursor
  E previous();            // Element before the cursor
  int nextIndex();         // Index of element after the cursor
  int previousIndex();     // Index of element before the cursor
  void remove();           // Optional
  void set(E o);           // Optional
  void add(E o);           // Optional
}
```

The `ListIterator<E>` interface is a bidirectional iterator for lists. It extends the `Iterator<E>` interface and allows the list to be iterated over in either direction. When iterating over lists, it can be helpful to imagine a *cursor* moving forward or backward *between* the elements when calls are made to the `next()` and the `previous()` methods, respectively. The element that the cursor passes over is returned. When the `remove()` method is called, the element last passed over is removed from the list.

The `ArrayList<E>`, `LinkedList<E>`, and `Vector<E>` Classes

Three implementations of the `List<E>` interface are provided in the `java.util` package: `ArrayList<E>`, `LinkedList<E>`, and `Vector<E>`.

The `ArrayList<E>` class implements the `List<E>` interface (Chapter 12, p. 643). The `Vector<E>` class is a legacy class that has been retrofitted to implement the `List<E>` interface, and will not be discussed in detail. The `Vector<E>` and `ArrayList<E>` classes are implemented using dynamically resizable arrays, providing fast *random access by index* (i.e., position-based access) and fast list iteration—very much like using an ordinary array. Unlike the `ArrayList<E>` class, the `Vector<E>` class is thread-safe, meaning that concurrent calls to the vector will not compromise its integrity. The `LinkedList<E>` implementation uses a doubly linked list. Insertions and deletions in a doubly linked list are very efficient.

The `ArrayList<E>` and `Vector<E>` classes offer comparable performance, but `Vector`s suffer a performance penalty due to synchronization. Position-based access has constant-time performance for the `ArrayList<E>` and `Vector<E>` classes. However, position-based access is in linear time for a `LinkedList<E>`, owing to iteration in a doubly linked list. When frequent insertions and deletions occur inside a list, a `LinkedList<E>` can be worth considering. In most cases, the `ArrayList<E>` implementation is the overall best choice for implementing lists.

In addition to the `List<E>` interface, the `LinkedList<E>` class also implements two other interfaces that allow it to be used for stacks and different kinds of queues (p. 814).

Example 15.3 illustrates some basic operations on lists. The user gets one shot at guessing a five-digit code. The solution is hardwired in the example as a list of five `Integer` objects. The `secretSolution` list is created at (1), and populated using the `Collections.addAll()` static method (p. 862). The guess specified at the command line is placed in a separate list, called `guess`, at (2).

The number of digits that are correctly guessed is determined at (3). The solution is first duplicated, and each digit in the guess is removed from the duplicated solution. The number of deletions corresponds to the number of correct digits in the `guess` list. A digit at a particular index in the guess list is returned by the `get()` method. The `remove()` method returns `true` if the duplicate list was modified—that is, the digit from the guess was found and removed from the duplicated solution. Of course, one could use the `retainAll()` method, as shown below, but the idea in Example 15.3 is to use positional access on the `guess` list.

Click here to view code image

```
// Find the number of digits that were correctly included.                     (3)
List<Integer> duplicate = new ArrayList<>(secretSolution);
duplicate.retainAll(guess);
numOfDigitsIncluded = duplicate.size();
```

Finding the number of digits that are correctly placed is achieved by using two list iterators at (4), which allow digits in the same position in the `guess` and the `secretSolution` lists to be compared.

Example 15.3 *Using Lists*

Click here to view code image

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

public class TakeAGuess {
  static final int NUM_DIGITS = 5;

  public static void main(String[] args) {

    // Sanity check on the given data.
    try {
      if (args.length != 1 || args[0].length() != NUM_DIGITS)
        throw new IllegalArgumentException();
      Integer.parseInt(args[0]);
    } catch(IllegalArgumentException nfe) {
      System.err.println("Guess should be " + NUM_DIGITS + " digits.");
      return;
    }
    String guessStr = args[0];
    System.out.println("Guess: " + guessStr);

    /* Initialize the solution list. This program has a fixed solution: 53272
    List<Integer> secretSolution = new ArrayList<>();                         //
    Collections.addAll(secretSolution, 5, 3, 2, 7, 2);

    // Convert the guess from string to a list of Integers.
    List<Integer> guess = new ArrayList<>();
    for (int i = 0; i < guessStr.length(); i++)
```

```
      guess.add(Character.getNumericValue(guessStr.charAt(i)));

    // Find the number of digits that were correctly included.
    List<Integer> duplicate = new ArrayList<>(secretSolution);
    int numOfDigitsIncluded = 0;
    for (int i = 0; i < NUM_DIGITS; i++)
      if (duplicate.remove(guess.get(i))) numOfDigitsIncluded++;

    /* Find the number of digits correctly placed by comparing the two
       lists, element by element, counting each correct placement. */
    // Need two iterators to traverse through the guess and solution lists.
    ListIterator<Integer> correct = secretSolution.listIterator();
    ListIterator<Integer> attempt = guess.listIterator();
    int numOfDigitsPlaced = 0;
    while (correct.hasNext())
      if (correct.next().equals(attempt.next())) numOfDigitsPlaced++;

    // Print the results.
    System.out.println(numOfDigitsIncluded + " digit(s) correctly included.");
    System.out.println(numOfDigitsPlaced +   " digit(s) correctly placed.");
  }
}
```

Running the program with the following command:

```
>java TakeAGuess 32227
```

gives the following output:

Click here to view code image

```
Guess: 32227
4 digit(s) correctly included.
1 digit(s) correctly placed.
```

15.4 Sets

The Java Collections Framework provides the `Set<E>` interface to model the mathematical *set* abstraction.

The `Set<E>` Interface

Unlike other implementations of the `Collection<E>` interface, implementations of the `Set<E>` interface do not allow duplicate elements. The `Set<E>` interface does not define any new methods, and its `add()` and `addAll()` methods will not store duplicates. If an element is not currently in the set, two consecutive calls to the `add()` method to insert the element will first return `true`, then `false`. A `Set<E>` models a mathematical set (see Table 15.3)—that is, it is an unordered collection of distinct objects.

Table 15.3 *Bulk Operations and Set Logic*

Set methods (a and b are sets)	Corresponding mathematical operations
`a.containsAll(b)`	$b \subseteq a$ (subset)
`a.addAll(b)`	$a = a \cup b$ (union)
`a.removeAll(b)`	$a = a - b$ (difference)
`a.retainAll(b)`	$a = a \cap b$ (intersection)
`a.clear()`	$a = \emptyset$ (empty set)

Multisets (also called *bags*) that allow duplicate elements cannot be implemented using the `Set<E>` interface, since this interface requires that elements are unique in the collection. An implementation of the `Set<E>` interface can choose to allow the `null` value, but the concrete class `TreeSet<E>` does *not*.

Creating Unmodifiable Sets

Unmodifiable collections are useful to prevent a collection from accidently being modified, as doing so might cause the program to behave incorrectly.

Creating and using unmodifiable sets is very similar to creating and using unmodifiable lists. Not surprisingly, the discussion here on unmodifiable sets reflects the discussion on unmodifiable lists (§12.2, p. 649). Later we will discuss unmodifiable maps (p. 832) and unmodifiable view collections (p. 856).

The `Set<E>` interface provides generic static methods to create *unmodifiable* sets that have the following characteristics:

- An unmodifiable set cannot be modified *structurally*; for example, elements cannot be added, removed, or replaced in such a set. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the elements themselves are mutable, the elements may appear modified.
- Both duplicates and `null` elements are not allowed, and will result in a `NullPointerException` if an attempt is made to create them with such elements.
- The order of the elements in such a set is unspecified.
- Such a set can be serialized if its elements are serializable (§20.5, p. 1261).

Click here to view code image

```
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5,
```

> ```
>                                    E e6, E e7, E e8, E e9, E e10)
> ```
>
> This method is overloaded, accepting any number of elements from 0 to 10. It returns an unmodifiable set containing the number of elements specified. It throws a `NullPointerException` if an element is `null`.
>
> Click here to view code image
>
> ```
> @SafeVarargs static <E> Set<E> of(E... elements)
> ```
>
> This variable arity method returns an unmodifiable set containing an arbitrary number of elements. It throws a `NullPointerException` if an element is `null` or if the array is `null`. The annotation suppresses the heap pollution warning in its declaration and unchecked generic array creation warning at the call sites.
>
> Click here to view code image
>
> ```
> static <E> Set<E> copyOf(Collection<? extends E> collection)
> ```
>
> This generic method returns an unmodifiable set containing the elements of the specified collection, in its iteration order. The specified collection must not be `null`, and it must not contain any `null` elements—otherwise, a `NullPointerException` is thrown. If the specified collection is subsequently modified, the returned set will not reflect such modifications.

The code below shows that a set created by the `Set.of()` method cannot be modified. The set returned is also not an instance of the class `HashSet`.

Click here to view code image

```
Set<String> set = Set.of("Tom", "Dick", "Harriet");
//  set.add("Harry");                           // UnsupportedOperationExceptio
//  set.remove(2);                              // UnsupportedOperationExceptio
System.out.println(set);                        // [Harriet, Tom, Dick]
System.out.println(set instanceof HashSet);     // false
```

The `Set.of()` method does not allow `null` elements:

Click here to view code image

```
Set<String> coinSet = Set.of("dime", "nickel", null); // NullPointerException
```

For arguments up to 10, an appropriate fixed-arity `Set.of()` method is called. Above 10 arguments, the variable arity `Set.of(E...)` method is called, passing an implicitly created array containing the arguments.

Click here to view code image

```
Set<Integer> intSet1 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);     // Fixed-ari
Set<Integer> intSet2 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Varargs
System.out.println(intSet1);        // [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
System.out.println(intSet2);        // [7, 6, 5, 4, 3, 2, 1, 11, 10, 9, 8]
```

At (1) below, an explicit array is passed as an argument, resulting in the variable arity `Set.of(E...)` method being called, creating a set of `String`. At (2), the method call explicitly specifies the type of its argument as `String[]`. In this case the one-argument `Set.of(E)` method is called, creating a set of size 1 and whose element type is `String[]`.

Click here to view code image

```
String[] strArray = {"Tom", "Dick", "Harriet"};
Set<String> strSet = Set.of(strArray);                     // (1) Set of String
Set<String[]> strArraySet = Set.<String[]>of(strArray);  // (2) Set of String[
System.out.println(strSet);                            // [Harriet, Dick, Tom]
System.out.println(strArraySet);                       // [[Ljava.lang.String;@3b22cdd0]
```

The code below shows how we can make a copy of a collection, in this case, a set. The `copyOf()` method creates a copy of the set passed as an argument at (1). The set created is unmodifiable analogous to the sets created with the `Set.of()` methods. The code also shows that modifying the original set does *not* reflect in the copy of the set.

Click here to view code image

```
Set<String> fab4 = new HashSet<>();
fab4.add("John"); fab4.add("Paul"); fab4.add("George"); fab4.add("Ringo");
System.out.println(fab4);                              // [George, John, Ringo, Paul]
Set<String> fabAlways = Set.copyOf(fab4);     // (1)
fab4.remove("John"); fab4.remove("George");  // Modify original set
System.out.println(fab4);                              // [Ringo, Paul]
System.out.println(fabAlways);                         // [John, Paul, Ringo, George]
```

The `HashSet<E>` and `LinkedHashSet<E>` Classes

The `HashSet<E>` class implements the `Set<E>` interface. Since this implementation uses a hash table, it offers near-constant-time performance for most operations. A `HashSet<E>` does not guarantee any ordering of the elements. However, the `LinkedHashSet<E>` subclass of `HashSet<E>` guarantees insertion order. It is also the implementation of choice if frequent iteration is necessary over the set. The sorted counterpart is `TreeSet<E>`, which implements the `SortedSet<E>` and the `NavigableSet<E>` interfaces and has logarithmic time complexity (p. 810).

A `HashSet<E>` relies on the implementation of the `hashCode()` and `equals()` methods of its elements (§14.2, p. 744, and §14.3, p. 753). The `hashCode()` method is used for hashing the elements, and the `equals()` method is needed for comparing elements for equality. In fact, the equality and the hash codes of `HashSet`s are defined in terms of the equality and the hash codes of their elements.

> `HashSet()`
>
> Constructs a new, empty set.
>
> Click here to view code image
>
> `HashSet(Collection<? extends E> c)`
>
> Constructs a new set containing the elements in the specified collection. The new set will not contain any duplicates. This offers a convenient way to remove duplicates from a collection.
>
> `HashSet(int initialCapacity)`
>
> Constructs a new, empty set with the specified initial capacity.
>
> Click here to view code image
>
> `HashSet(int initialCapacity, float loadFactor)`
>
> Constructs a new, empty set with the specified initial capacity and the specified load factor.

As mentioned earlier, the `LinkedHashSet<E>` implementation is a subclass of the `HashSet<E>` class. It works similarly to a `HashSet<E>` except for one important detail. Unlike a `HashSet<E>`, a `LinkedHashSet<E>` guarantees that the iterator will access the elements in *insertion order*—that is, in the order in which the elements were inserted into the `LinkedHashSet<E>`.

The `LinkedHashSet<E>` class offers constructors analogous to the ones in the `Hash-Set<E>` class. The initial *capacity* (i.e., the number of buckets in the hash table) and its *load factor* (i.e., the ratio of number of elements stored to its current capacity) can be tuned when the set is created. The default values for these parameters will, under most circumstances, provide acceptable performance.

Example 15.4 demonstrates iterating over a `HashSet<E>` and a `LinkedHashSet<E>` created at (1) and (2), respectively. Regardless of the order in which elements are inserted into a `HashSet<E>`, we cannot depend on the order in which the `for(:)` loop will iterate over the elements in the set, as is evident from the program output.

A `LinkedHashSet<E>`, on the other hand, is always iterated over in *insertion order*—that is, the first element inserted is the first element retrieved by the `for(:)` loop. The program output confirms this behavior, as the meal that was inserted last into the `LinkedHashSet<E>` is served first. The same behavior will be exhibited if an explicit iterator is used to iterate over the sets.

Example 15.4 *Iterating Over Sets*

Click here to view code image

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
public class IterationHashSetAndLinkedHashSet {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<>();                            // (1)
    set1.add("breakfast"); set1.add("lunch"); set1.add("dinner");
    System.out.println("Serving meals from a HashSet (order can vary):");
    for (String meal : set1) {
      System.out.println(meal);
    }
    Set<String> set2 = new LinkedHashSet<>();                      // (2)
    set2.add("breakfast"); set2.add("lunch"); set2.add("dinner");
    System.out.println("Serving meals from a LinkedHashSet" +
                       " (always insertion order):");
    for (String meal : set2) {
      System.out.println(meal);
    }
  }
}
```

Output from the program:

Click here to view code image

```
Serving meals from a HashSet (order can vary):
dinner
breakfast
lunch
Serving meals from a LinkedHashSet (always insertion order):
breakfast
lunch
dinner
```

Example 15.5 demonstrates set operations. It determines the following relationships between two sets of characters:

- Whether they are disjunct—that is, have no elements in common
- Whether they have the same elements—that is, are equivalent
- Whether one is a subset of the other

- Whether one is a superset of the other
- Whether they have a common subset

Given a list of words as program arguments, each argument is turned into a set of characters. This character set is compared with the set of all characters encountered so far in previous arguments.

The set `encountered` created at (1) accumulates characters as each argument is processed. For each argument, an empty set of characters is created at (2). This `characters` set is populated with the characters of the current argument at (3). The program first determines if there is a common subset between the two sets at (4)—that is, whether the current argument has any characters that were in previous arguments:

Click here to view code image

```
// Determine whether a common subset exists:                   (4)
Set<Character> commonSubset = new HashSet<>(encountered);
commonSubset.retainAll(characters);
boolean areDisjunct = commonSubset.size()==0;
```

Note that the `retainAll()` operation is destructive. The code at (4) does not affect the `encountered` and the `characters` sets. If the size of the common subset is zero, the sets are disjunct; otherwise, the relationship must be narrowed down. The subset and superset relations are determined at (5), using the `containsAll()` method.

Click here to view code image

```
// Determine superset and subset relations.                   (5)
boolean isSubset = encountered.containsAll(characters);
boolean isSuperset = characters.containsAll(encountered);
```

The sets are equivalent if both of the previous relations are `true`. If the relations are both `false`—that is, no subset or superset relationship exists, the sets only have the subset computed at (4) in common. The `encountered` set is updated with the contents of the `characters` set to accumulate all characters encountered so far. The `addAll()` method is used for this purpose at (6):

Click here to view code image

```
encountered.addAll(characters);                              // (6)
```

As we can see from the output, the program prints the contents of the sets in the standard text representation for collections.

Example 15.5 *Using Sets*

Click here to view code image

```
import java.util.HashSet;
import java.util.Set;

public class CharacterSets {
  public static void main(String[] args) {

    // A set for keeping track of all characters previously encountered.
    Set<Character> encountered = new HashSet<>();               // (1)

    // For each program argument in the command line ...
    for (String argument : args) {

      // Convert the current argument to a set of characters.
      Set<Character> characters = new HashSet<>();              // (2)
      int size = argument.length();
      // For each character in the argument...
      for (int j = 0; j < size; j++)
        // add character to the characters set.
        characters.add(argument.charAt(j));                     // (3)

      // Determine whether a common subset exists:                 (4)
      Set<Character> commonSubset = new HashSet<>(encountered);
      commonSubset.retainAll(characters);
      boolean areDisjunct = commonSubset.size()==0;

      if (areDisjunct) {
        System.out.println(characters + " and " + encountered + " are disjunct
      } else {
        // Determine superset and subset relations.                (5)
        boolean isSubset = encountered.containsAll(characters);
        boolean isSuperset = characters.containsAll(encountered);
        if (isSubset && isSuperset)
          System.out.println(characters + " is equivalent to " + encountered);
        else if (isSubset)
          System.out.println(characters + " is a subset of " + encountered);
        else if (isSuperset)
          System.out.println(characters + " is a superset of " + encountered);
        else
          System.out.println(characters + " and " + encountered + " have " +
                             commonSubset + " in common.");
      }

      // Update the set of characters encountered so far.
      encountered.addAll(characters);                           // (6)
    }
  }
}
```

Running the program with the following arguments:

Click here to view code image

```
>java CharacterSets i said i am maids
```

results in the following output:

Click here to view code image

```
[i] and [] are disjunct.
[d, a, s, i] is a superset of [i]
[i] is a subset of [d, a, s, i]
[a, m] and [d, a, s, i] have [a] in common.
[d, a, s, m, i] is equivalent to [d, a, s, m, i]
```

15.5 Sorted Sets and Navigable Sets

Before reading this subsection, it is a good idea to review the `Comparable<E>` interface (§14.4, p. 761) for defining the natural ordering for objects, and the `Comparator<E>` interface (§14.5, p. 769) for defining a particular total ordering for objects.

The `SortedSet<E>` Interface

The `SortedSet<E>` interface extends the `Set<E>` interface to provide the functionality for handling sorted sets. Since the elements are sorted, iterating over the set using either the `for(:)` loop or an iterator will access the elements according to the ordering used by the set.

```
// First-last elements
E first()
E last()
```

> The `first()` method returns the first element currently in this sorted set, and the `last()` method returns the last element currently in this sorted set. The elements are chosen based on the ordering used by the sorted set. Both throw a `NoSuchElementException` if the sorted set is empty.

Click here to view code image

```
// Range-view operations
SortedSet<E> headSet(<E> toElement)
SortedSet<E> tailSet(<E> fromElement)
SortedSet<E> subSet(<E> fromElement, <E> toElement)
```

> The `headSet()` method returns a *view* of a portion of this sorted set, whose elements are strictly less than the specified element. Similarly, the `tailSet()` method returns a view of the portion of this sorted set, whose elements are greater than or equal to the specified element. The `subSet()` method returns a view of the portion of this sorted set, whose elements range from `fromElement`, inclusive, to `toElement`, exclusive (also called *half-open interval*). It throws an

> `IllegalArgumentException` if the `fromElement` is greater than the `toElement`.
>
> Note that the views present the elements sorted in the same order as the underlying sorted set. Also, changes made through views are reflected in the underlying sorted set, and vice versa.
>
> Click here to view code image
>
> ```
> // Comparator access
> Comparator<? super E> comparator()
> ```
>
> Returns the comparator associated with this sorted set, or `null` if it uses the natural ordering of its elements. This comparator, if defined, is used by default when a sorted set is constructed, and used when copying elements into new sorted sets.

The `NavigableSet<E>` Interface

The `NavigableSet<E>` interface extends the `SortedSet<E>` interface with *navigation methods* to find the closest matches for given search targets. By navigation we mean operations that require searching for elements in the navigable set. In the absence of elements, these operations return `null` rather than throwing a `NoSuchElementException`, as is the case for the methods in the `SortedSet<E>` interface.

The `NavigableSet<E>` interface replaces the `SortedSet<E>` interface and is the preferred choice when a sorted set is required. In addition to the methods of the `Sorted-Set<E>` interface, the `NavigableSet<E>` interface adds the following *new* methods:

> ```
> // First-last elements
> E pollFirst()
> E pollLast()
> ```
>
> The `pollFirst()` method removes and returns the first element and the `poll-Last()` method removes and returns the last element currently in this navigable set. The element is determined according to some policy employed by the set—for example, queue policy. Both return `null` if the sorted set is empty.
>
> Click here to view code image
>
> ```
> // Range-view operations
> NavigableSet<E> headSet(E toElement,   boolean inclusive)
> NavigableSet<E> tailSet(E fromElement, boolean inclusive)
> ```

```
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

These operations are analogous to the ones in the `SortedSet<E>` interface (p. 810), returning different views of the underlying navigable set, depending on the bound elements. However, the bound elements can be *excluded or included* by the operation, depending on the value of the `boolean` argument `inclusive`.

```
// Closest-matches
E ceiling(E e)
E floor(E e)
E higher(E e)
E lower(E e)
```

The method `ceiling()` returns the least element in the navigable set greater than or equal to argument `e`. The method `floor()` returns the greatest element in the navigable set less than or equal to argument `e`. The method `higher()` returns the least element in the navigable set strictly greater than argument `e`. The method `lower()` returns the greatest element in the navigable set strictly less than argument `e`. All methods return `null` if the required element is not found.

Click here to view code image

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

The first method returns a reverse-order iterator for the navigable set. The second method returns a reverse-order view of the elements in the navigable set.

The `TreeSet<E>` Class

The `TreeSet<E>` class implements the `NavigableSet<E>` interface and thereby the `SortedSet<E>` interface. By default, operations on a sorted set rely on the natural ordering of the elements. However, a total ordering can be specified by passing a customized comparator to the constructor.

The `TreeSet<E>` class maintains non-`null` elements in sort order, and iteration is also in the same sort order.

The `TreeSet<E>` implementation uses balanced trees, which deliver excellent (logarithmic) performance for all operations. However, searching in a `HashSet<E>` can be faster than in a `TreeSet<E>`

because hashing algorithms usually offer better performance than the search algorithms for balanced trees. The `TreeSet<E>` class is preferred if elements are to be maintained in sort order and if fast insertion and retrieval of individual elements is desired.

The `TreeSet<E>` class provides four constructors:

> ```
> TreeSet()
> ```
>
> The default constructor that creates a new, empty sorted set, according to the natural ordering of the elements.
>
> Click here to view code image
>
> ```
> TreeSet(Comparator<? super E> comparator)
> ```
>
> A constructor that takes an explicit comparator for specific total ordering of the elements.
>
> Click here to view code image
>
> ```
> TreeSet(Collection<? extends E> collection)
> ```
>
> A constructor that creates a sorted set based on a collection, according to the natural ordering of the elements.
>
> ```
> TreeSet(SortedSet<E> set)
> ```
>
> A constructor that creates a new set containing the same elements as the specified sorted set, with the same ordering.

Example 15.6 illustrates some selected navigation operations on a `TreeSet<E>`. Keep in mind that the Unicode values of uppercase letters are less than the Unicode values of lowercase letters—that is, the former occur before the latter in the Unicode standard.

The set is created at (1), and populated by calling the `Collections.addAll()` method at (2). The elements are maintained according to the natural ordering for `String`s— that is, the one defined by the `compareTo()` method of the `Comparable<String>` interface implemented by the `String` class. The subset-view operations at (3) show how the bounds can be inclusive or exclusive. Note also how the closest-match methods at (4) behave. A sorted set with the reverse order corresponding to the natural ordering is created at (5). The methods `pollFirst()` and `pollLast()` remove the element that is retrieved—that is, they change the set structurally.

The following code shows how we can create a sorted set with a specific total ordering, by supplying a comparator in the constructor call:

Click here to view code image

```
  NavigableSet<String> strSetB = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
  Collections.addAll(strSetB, "strictly", "dancing", "Java", "Ballroom");
  System.out.println(strSetB);          // [Ballroom, dancing, Java, strictly]
```

Example 15.6 *Using Navigable Sets*

Click here to view code image

```
import java.util.Collections;
import java.util.NavigableSet;
import java.util.TreeSet;
import static java.lang.System.out;

public class SetNavigation {
  public static void main(String[] args) {

    NavigableSet<String> strSetA = new TreeSet<>();                          //
    Collections.addAll(strSetA, "Strictly", "Java", "dancing", "ballroom"); //
    out.println("Before: " + strSetA);      // [Java, Strictly, ballroom, danc

    out.println("\nSubset-views:");                      // (3)
    out.println(strSetA.headSet("ballroom", true));   // [Java, Strictly, ballr
    out.println(strSetA.headSet("ballroom", false)); // [Java, Strictly]
    out.println(strSetA.tailSet("Strictly", true));   // [Strictly, ballroom,
                                                       //  dancing]
    out.println(strSetA.tailSet("Strictly", false)); // [ballroom, dancing]
    out.println(strSetA.subSet("A", false, "Z", false )); // [Java, Strictly]
    out.println(strSetA.subSet("a", false, "z", false )); // [ballroom, dancin

    out.println("\nClosest-matches:");                   // (4)
    out.println(strSetA.ceiling("ball"));               // ballroom
    out.println(strSetA.floor("ball"));                 // Strictly
    out.println(strSetA.higher("ballroom"));            // dancing
    out.println(strSetA.lower("ballroom"));             // Strictly

    out.println("\nReverse order:");                     // (5)
    out.println(strSetA.descendingSet());   // [dancing, ballroom, Strictly, J

    out.println("\nFirst-last elements:");
    out.println(strSetA.pollFirst());                   // Java
    out.println(strSetA.pollLast());                    // dancing

    out.println("\nAfter: " + strSetA);                  // [Strictly, ballroom]
  }
}
```

Output from the program:

Click here to view code image

```
Before: [Java, Strictly, ballroom, dancing]

Subset-views:
[Java, Strictly, ballroom]
[Java, Strictly]
[Strictly, ballroom, dancing]
```

```
[ballroom, dancing]
[Java, Strictly]
[ballroom, dancing]

Closest-matches:
ballroom
Strictly
dancing
Strictly

Reverse order:
[dancing, ballroom, Strictly, Java]

First-last elements:
Java
dancing

After: [Strictly, ballroom]
```

15.6 Queues

In this section we look at the different types of queues provided by the Java Collections Framework.

The `Queue<E>` Interface

The `Queue<E>` interface extends the `Collection<E>` interface to specify a general contract for queues. A *queue* is a collection that maintains elements in *processing order*. An implementation of the `Queue<E>` interface provides the queue policy for yielding the next element for processing. A *head* position in the queue specifies where the next element for processing can be obtained. A basic queue usually maintains its elements in FIFO (*first in, first out*) ordering, but other orderings are also quite common: LIFO (*last in, first out*) ordering (also called *stacks*) and priority ordering (also called *priority queues*). The order in which elements of a queue can be retrieved for processing is dictated either by the natural ordering of the elements or by a comparator. A queue can be *unbounded* or *capacity-restricted*, depending on its implementation.

The `Queue<E>` interface extends the `Collection<E>` interface with the following methods. A summary of these methods is presented in Table 15.4.

Table 15.4 *Summary of Methods in the Queue Interface*

Operation	Throws exception	Returns special value
Insert at the tail	`add(e)` *can throw* `IllegalArgumentException`	`offer(e)` *returns* `true` *or* `false`
Remove from the head	`remove()` *throws* `NoSuchElementException`	`poll()` *returns head element or* `null`
Examine element at the head	`element()` *throws* `NoSuchElementException`	`peek()` *returns head element or* `null`

`NoSuchElementException`

```
// Insert
boolean add(E element)
boolean offer(E element)
```

Both methods insert the specified element in the queue. The return value indicates the success or failure of the operation. The `add()` method inherited from the `Collection<E>` interface throws an `IllegalStateException` if the queue is full, but the `offer()` method does not.

```
// Remove
E remove()
E poll()
```

Both methods retrieve the head element and remove it from the queue. If the queue is empty, the `remove()` method throws a `NoSuchElementException`, but the `poll()` method returns the `null` value.

```
// Examine
E element()
E peek()
```

Both methods retrieve the head element, but do *not* remove it from the queue. If the queue is empty, the `element()` method throws a `NoSuchElementException`, but the `peek()` method returns the `null` value.

The `PriorityQueue<E>` and `LinkedList<E>` Classes

Both the `PriorityQueue<E>` and `LinkedList<E>` classes implement the `Queue<E>` interface. Unless bidirectional iteration is necessary, other queue implementations should be considered, and not the `LinkedList<E>` class. (The `LinkedList<E>` class is also eclipsed by the introduction of the `ArrayDeque<E>` class when it comes to implementing deques, as we will see shortly.)

As the name suggests, the `PriorityQueue<E>` class is the obvious implementation for a queue with priority ordering. The implementation is based on a *priority heap*, a tree-like structure that yields an element at the head of the queue according to the priority ordering, which is defined either by the natural ordering of its elements or by a comparator. In the case of several elements having the same priority, one of them is chosen arbitrarily.

Elements of a `PriorityQueue<E>` are *not* sorted. The queue only guarantees that elements can be *removed* in priority order, and any iteration using an iterator does *not* guarantee to abide by the priority order.

The `PriorityQueue<E>` class provides the following constructors:

> Click here to view code image
>
> ```
> PriorityQueue()
> PriorityQueue(int initialCapacity)
> ```
>
> The default constructor creates a new, empty `PriorityQueue` with default initial capacity and natural ordering. The second constructor creates a new, empty `PriorityQueue` with the specified initial capacity and natural ordering.
>
> Click here to view code image
>
> ```
> PriorityQueue(Comparator<? super E> comparator)
> PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
> ```
>
> Both constructors create a new, empty `PriorityQueue` where the ordering is defined by the specified `comparator`. The priority queue created by the first and the second constructors has the default initial capacity and the specified initial capacity, respectively.
>
> Click here to view code image
>
> ```
> PriorityQueue(PriorityQueue<? extends E> pq)
> PriorityQueue(SortedSet<? extends E> set)
> PriorityQueue(Collection<? extends E> c)
> ```
>
> The first and the second constructors create a new `PriorityQueue` with the ordering and the elements from the specified priority queue or sorted set, respectively.
>
> The last constructor creates a new `PriorityQueue` containing the elements in the specified collection. It will have natural ordering of its elements, unless the specified collection is either a `SortedSet<E>` or another `PriorityQueue`, in which case, the collection's ordering will be used.

Example 15.7 illustrates using priority queues. The example shows how priority queues maintain objects of the `Task` class. The natural ordering of the objects in this class is based on the *task number* (`Integer`). This natural ordering will result in the priority queue yielding its elements in *ascending* order of the task numbers—that is, tasks with *smaller* task numbers will have *higher* priority.

In Example 15.7, the `main()` method in the class `TaskExecutor` creates an array with some tasks at (1). The method essentially creates empty priority queues with different priority orderings, at (2) through (7), and calls the `testPQ()` method at (8) to execute tasks passed in the array using the supplied priority queue.

Click here to view code image

```
private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {    // (
...
}
```

The `testPQ()` method at (8) loads the queue at (9) from the array of tasks. It calls the `offer()` method to insert a task in the priority queue. The method then calls the `peek()` method at (10) to examine the task at the head of the queue. The tasks are executed by removing them one by one at (11) by calling the `poll()` method. The output shows the order in which the tasks are executed, depending on the priority ordering.

The priority queue `pq1` at (2) has its priority ordering defined by the natural ordering of the tasks.

Click here to view code image

```
PriorityQueue<Task> pq1 = new PriorityQueue<>();                          // (2)
```

Note that the text representation of the queue in the output

Click here to view code image

```
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
```

does *not* reflect the tasks in priority order. It just shows what tasks are in the queue. The text representation of the queue is generated by the print method running an iterator over the queue. The iterator is under no obligation to take the priority order into consideration. The output also shows that the task with the highest priority (i.e., the smallest task number) is at the head of the queue:

Click here to view code image

```
Task at the head: 100@breakfast
```

The call to the `poll()` method in the `while` loop at (11) removes tasks in priority order, as verified by the output:

Click here to view code image

```
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner
```

Since two of the tasks have the same priority, the queue selects which one should be chosen first. The queue is empty when the `peek()` method returns `null`.

The priority queue `pq2` at (3) has its priority ordering defined by the reverse natural ordering returned by the static method `reverseOrder()` of the `Comparator<E>` interface:

Click here to view code image

```
PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
```

Both priority queues `pq3` and `pq4` use reversed ordering based on the *task name*. This ordering is defined for `pq3` and `pq4` by a lambda expression at (4) and by methods of the `Comparator<E>` interface at (7), respectively. The latter implementation is obviously to be preferred:

Click here to view code image

```
PriorityQueue<Task> pq4 = new PriorityQueue<>(                                    // (5)
    Comparator.comparing(Task::getTaskName).reversed()
);
```

The static method `comparing()` extracts a comparator that uses the `getName()` method of the `Task` class, and the default method `reversed()` reverses this comparator.

The priority queues `pq5` and `pq6` use total ordering based on multiple fields of the `Task` class: on the task number, followed by the task name. This ordering is defined for `pq5` and `pq6` by a lambda expression at (6) and by methods of the `Comparator<E>` interface at (7), respectively. The latter implementation first extracts a comparator based on the task number, and chains it with one based on the task name:

Click here to view code image

```
PriorityQueue<Task> pq6 = new PriorityQueue<>(                                    // (7)
      Comparator.comparing(Task::getTaskNumber)
                .thenComparing(Task::getTaskName)
);
```

We leave it to the reader to verify that the output conforms to the priority ordering of priority queues at (3) through (7).

Example 15.7 *Using Priority Queues*

Click here to view code image

```
/** Represents a task. */
public class Task implements Comparable<Task> {
  private Integer taskNumber;
  private String  taskName;

  public Task(Integer tp, String tn) {
    taskNumber = tp;
    taskName   = tn;
  }

  @Override
  public boolean equals(Object obj) {// Equality based on the task number.
    return (this == obj)
        || (obj instanceof Task other
            && this.taskNumber.equals(other.taskNumber));
  }
  @Override
  public int compareTo(Task task2) { // Natural ordering based on the task num
    return this.taskNumber.compareTo(task2.taskNumber);
  }
  @Override
  public int hashCode() {            // Hash code based on the task number.
    return this.taskNumber.hashCode();
  }
  @Override
  public String toString() { return taskNumber + "@" + taskName; }

  public String  getTaskName()   { return taskName; }
  public Integer getTaskNumber() { return taskNumber;}
}
```

Click here to view code image

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;
import static java.lang.System.out;
/** Executes tasks. */
public class TaskExecutor {

  public static void main(String[] args) {
    // Array with some tasks.
    Task[] taskArray = {
        new Task(200, "lunch"), new Task(200, "tea"),
        new Task(300, "dinner"), new Task(100, "breakfast"),
    };
    out.println("Array of tasks: " + Arrays.toString(taskArray));

    out.println("Priority queue using natural ordering (task number).");
    PriorityQueue<Task> pq1 = new PriorityQueue<>();                        //
    testPQ(taskArray, pq1);

    out.println("Priority queue using reverse natural ordering.");      //
    PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
    testPQ(taskArray, pq2);

    out.println("Priority queue using reversed ordering"
        + " on task name (lambda expression).");
    PriorityQueue<Task> pq3 = new PriorityQueue<>(                          //
        (task1, task2) -> {
          String taskName1 = task1.getTaskName();
          String taskName2 = task2.getTaskName();
          return -taskName1.compareTo(taskName2);
         }
      );
    testPQ(taskArray, pq3);

    out.println("Priority queue using reversed ordering"
        + " on task name (extracted comparator).");
    PriorityQueue<Task> pq4 = new PriorityQueue<>(                         //
        Comparator.comparing(Task::getTaskName).reversed()
    );
    testPQ(taskArray, pq4);

    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (lambda expression).");
    PriorityQueue<Task> pq5 = new PriorityQueue<>(                         //
      (task1, task2) -> {
        Integer taskNumber1 = task1.getTaskNumber();
        Integer taskNumber2 = task2.getTaskNumber();
        if (!taskNumber1.equals(taskNumber2))
          return taskNumber1.compareTo(taskNumber2);
        String taskName1 = task1.getTaskName();
        String taskName2 = task2.getTaskName();
        if (!taskName1.equals(taskName2))
          return taskName1.compareTo(taskName2);
        return 0;
       }
    );
    testPQ(taskArray, pq5);

    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (extracted comparators).");
    PriorityQueue<Task> pq6 = new PriorityQueue<>(                         //
        Comparator.comparing(Task::getTaskNumber)
                  .thenComparing(Task::getTaskName)
    );
    testPQ(taskArray, pq6);
  }

  // Runs tasks.
  private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {   //
    // Load the tasks:
    for (Task task : taskArray)
      pq.offer(task);
    out.println("Queue before executing tasks: " + pq);

    // Peek at the head:
    out.println("Task at the head: " + pq.peek());

    // Do the tasks:
    out.print("Doing tasks: ");
    while (!pq.isEmpty()) {
      Task task = pq.poll();
      out.print(task + " ");
    }
    out.println();
    out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Array of tasks: [200@lunch, 200@tea, 300@dinner, 100@breakfast]
Priority queue using natural ordering (task number).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner
```

```
Priority queue using reverse natural ordering.
Queue before executing tasks: [300@dinner, 200@tea, 200@lunch, 100@breakfast]
Task at the head: 300@dinner
Doing tasks: 300@dinner 200@tea 200@lunch 100@breakfast

Priority queue using reversed ordering on task name (lambda expression).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using reversed ordering on task name (extracted comparator).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using total ordering based on task number,
followed by task name (lambda expression).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner

Priority queue using total ordering based on task number,
followed by task name (extracted comparators).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner
```

15.7 Deques

In this section we look at *deques*—that is, linear collections that allow processing of elements from both ends.

The `Deque<E>` Interface

The `Deque<E>` interface extends the `Queue<E>` interface to allow *double-ended queues*. Such a queue is called a *deque*. It allows operations not just at its *head* as a queue, but also at its *tail*. It is a linear unbounded structure in which elements can be inserted at or removed from *either* end. Various synonyms are used in the literature for the head and tail of a deque: front and back, first and last, start and end.

A deque can be used as a *FIFO queue*, where elements added at the tail are presented at the head for inspection or removal in the same order, thus implementing FIFO ordering. A deque can also be used as a *stack*, where elements are added to and removed from the *same* end, thus implementing LIFO ordering.

The `Deque<E>` interface defines symmetrical operations at its head and tail. Which end is in question is made evident by the method name. A *XXX*`First()` method and a *XXX*`Last()` method process an element at the *head* and at the *tail*, respectively. Below, equivalent methods from the `Queue<E>` interface are also identified. The `push()` and `pop()` methods are convenient for implementing stacks.

Click here to view code image

```
// Insert
boolean offerFirst(E element)
boolean offerLast(E element)                      Queue equivalent: offer()
void addFirst(E element)
void addLast(E element)                           Queue equivalent: add()
void push(E element)                              Synonym: addFirst()
```

Insert the specified element in the deque. They all throw a `NullPointerException` if the specified element is `null`. The `addFirst()` and `addLast()` methods throw an `IllegalStateException` if the element cannot be added, but the `offerFirst()` and `offerLast()` methods do not.

Click here to view code image

```
// Remove
E removeFirst()                                   Queue equivalent: remove()
E removeLast()
E pollFirst()                                     Queue equivalent: poll()
E pollLast()
E pop()                                           Synonym: removeFirst()
boolean removeFirstOccurence(Object obj)
boolean removeLastOccurence(Object obj)
```

Remove an element from the deque. The `removeFirst()` and `removeLast()` methods throw a `NoSuchElementException` if the deque is empty. The `pollFirst()` and `pollLast()` methods return `null` if the deque is empty.

Click here to view code image

```
// Examine
E getFirst()                                      Queue equivalent: element()
E getLast()
E peekFirst()                                     Queue equivalent: peek()
E peekLast()
```

Retrieve an element from the deque, but do not remove it from the deque. The `getFirst()` and `getLast()` methods throw a `NoSuchElementException` if the deque is empty. The `peekFirst()` and `peekLast()` methods return `null` if the deque is empty.

Click here to view code image

```
// Misc.
Iterator<E> descendingIterator()
```

Returns an iterator to iterate over the deque in reverse order—that is, from the tail to the head.

Table 15.5 summarizes the methods provided by the `Deque<E>` interface, showing which operations can be performed at the head and which ones at the tail of the deque. Each row indicates the runtime behavior of the methods in that row, whether they throw an exception or not. Any method whose name starts with either `"offer"`, `"poll"`, or `"peek"` does not throw an exception. Counterpart methods inherited from the `Queue<E>` interface are marked by an asterisk (*) in the table.

Table 15.5 *Summary of Deque Methods*

Insert at the head	Insert at the tail	Runtime behavior on failure
`offerFirst()`	`offerLast()`, `offer()*`	*Returns* `false` *if full*
`addFirst()`	`addLast()`, `add()*`	*Throws* `IllegalStateException`
Remove from the head	**Remove from the tail**	**Runtime behavior on failure**
`pollFirst()`	`pollLast()`, `poll()*`	*Returns* `null` *if empty*
`removeFirst()`	`removeLast()`, `remove()*`	*Throws* `NoSuchElementException`
`peekFirst()`	`peekLast()`, `peek()*`	*Returns* `null` *if empty*
`getFirst()`	`getLast()`, `element()*`	*Throws* `NoSuchElementException`

The `ArrayDeque<E>` and `LinkedList<E>` Classes

The `ArrayDeque<E>` and `LinkedList<E>` classes implement the `Deque<E>` interface. The `ArrayDeque<E>` class provides better performance than the `LinkedList<E>` class for implementing FIFO queues, and is also a better choice than the `java.util.Stack` class for implementing stacks.

An `ArrayDeque<E>` is also an `Iterable<E>`, and iteration is always from the head to the tail. The class provides the `descendingIterator()` method for iterating in reverse order. Since deques are not lists, positional access is not possible, nor can they be sorted. They are also not thread-safe, and `null` values are not allowed as elements.

The `ArrayDeque<E>` class provides the following constructors, analogous to the ones in the `ArrayList<E>` class:

```
ArrayDeque()
ArrayDeque(int numOfElements)
```

> The first constructor creates a new, empty `ArrayDeque` with an initial capacity to hold 16 elements.
>
> The second constructor creates a new, empty `ArrayDeque` with initial capacity required to hold the specified number of elements.
>
> Click here to view code image
>
> ```
> ArrayDeque(Collection<? extends E> c)
> ```

> Creates a new `ArrayDeque` containing the elements in the specified collection. The ordering in the `ArrayDeque` is determined by the iteration order of the iterator for the collection passed as an argument.
>
> The `LinkedList<E>` class provides constructors that are analogous to the first and the last constructors.

Example 15.8 illustrates the methods for inserting, examining, and removing elements from a deque. The program output shows how each method affects the deque when inserting, examining, and removing elements from either the head or the tail of a deque.

Example 15.8 *Demonstrating Deque Operations*

Click here to view code image

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeOperations {
  public static void main(String[] args) {
    Deque<String> deque = new ArrayDeque<String>();
    System.out.println("After creating the deque: " + deque);

    // Insert elements:
    deque.offerFirst("A (H)");                    // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.offerLast("B (T)");                     // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);
    deque.push("C (H)");                          // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addFirst("D (H)");                      // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addLast("E (T)");                       // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);

    // Examine element:
    System.out.println("Examine at the head: " + deque.getFirst());
    System.out.println("Examine at the tail: " + deque.getLast());
    System.out.println("Examine at the head: " + deque.peekFirst());
    System.out.println("Examine at the tail: " + deque.peekLast());

    // Remove elements:
    deque.removeFirst();                          // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.removeLast();                           // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pollFirst();                            // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.pollLast();                             // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pop();                                  // Remove from the head
    System.out.println("After removing from the head: " + deque);
  }
}
```

Output from the program:

Click here to view code image

```
After creating the deque: []
After inserting at the head: [A (H)]
After inserting at the tail: [A (H), B (T)]
After inserting at the head: [C (H), A (H), B (T)]
After inserting at the head: [D (H), C (H), A (H), B (T)]
After inserting at the tail: [D (H), C (H), A (H), B (T), E (T)]
Examine at the head: D (H)
Examine at the tail: E (T)
Examine at the head: D (H)
Examine at the tail: E (T)
After removing from the head: [C (H), A (H), B (T), E (T)]
After removing from the tail: [C (H), A (H), B (T)]
After removing from the head: [A (H), B (T)]
After removing from the tail: [A (H)]
After removing from the head: []
```

Example 15.9 illustrates using an `ArrayDeque` both as a LIFO stack and as a FIFO queue. Elements from an array are pushed onto the stack at (3), and then popped from the stack at (5). The call to the `isEmpty()` method in the `while` loop at (4) determines whether the stack is empty. The output shows that the elements were popped in the reverse order to the order in which they were inserted—that is, LIFO ordering.

Similarly, elements from an array are inserted at the tail of a FIFO queue at (8), and then removed from the head of the FIFO queue at (10). The call to the `isEmpty()` method in the `while` loop at (4) determines whether the FIFO queue is empty. The output shows that the elements were removed in the same order they were inserted—that is, FIFO ordering.

Note that in Example 15.9 the stack grows at the head of the deque, but the FIFO queue grows at the tail of the deque.

Example 15.9 *Using Deques as a LIFO Stack and as a FIFO Queue*

Click here to view code image

```
import java.util.ArrayDeque;
import java.util.Arrays;

/** Executes tasks. */
public class TaskExecutor2 {

  public static void main(String[] args) {
    String[] elementArray = {"sway", "and", "twist", "stacks", "tall"};     //
    System.out.println("Array of elements: " + Arrays.toString(elementArray));

    // Using ArrayDeque as a stack:
    ArrayDeque<String> stack = new ArrayDeque<>();
    for (String string : elementArray)
      stack.push(string);                                      // (3) Push elements.
    System.out.println("Stack before: TOP->" + stack + "<-BOTTOM");
    System.out.print("Popping stack: ");
    while (!stack.isEmpty()) {                        // (4)
      System.out.print(stack.pop() + " ");            // (5) Pop elements.
    }
    System.out.println("\n");

    // Using ArrayDeque as a FIFO queue:                 (6)
    elementArray = new String[] {"Waiting", "in", "queues", "is", "boring"};//
    System.out.println("Array of elements: " + Arrays.toString(elementArray));
    ArrayDeque<String> fifoQueue = new ArrayDeque<>();
    for (String string : elementArray)
      fifoQueue.offerLast(string);                    // (8) Insert at tail.
    System.out.println("Queue before: HEAD->" + fifoQueue  + "<-TAIL");
    System.out.print("Polling queue: ");
    while (!fifoQueue.isEmpty()) {                    // (9)
      String string = fifoQueue.pollFirst();          // (10) Remove from head
      System.out.print(string.toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Array of elements: [sway, and, twist, stacks, tall]
Stack before: TOP->[tall, stacks, twist, and, sway]<-BOTTOM
Popping stack: tall stacks twist and sway

Array of elements: [Waiting, in, queues, is, boring]
Queue before: HEAD->[Waiting, in, queues, is, boring]<-TAIL
Polling queue: WAITING IN QUEUES IS BORING
```

Review Questions

15.1 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ400A100 {
  public static void main(String[] args) {
    int sum = 0;
    for (int i : makeCollection()) {
      sum += i;
    }
    System.out.println(sum);
  }

  static Collection<Integer> makeCollection() {
    System.out.println("A collection coming up.");
    Collection<Integer> collection = new ArrayList<>();
    collection.add(10); collection.add(20); collection.add(30);
    return collection;
  }
}
```

Select the one correct answer.

a. The program will print

```
A collection coming up.
60
```

b. The program will print

```
A collection coming up.
A collection coming up.
A collection coming up. 60
```

c. The program will fail to compile.

d. None of the above

15.2 Given the following code:

Click here to view code image

```
import java.util.*;
class Fruity {
  private String fName;
  Fruity(String fName) { this.fName = fName; }

  public void setName(String newName) { this.fName = newName; }
  public String toString() { return fName; }
  public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof Fruity)) return false;
    return fName.equalsIgnoreCase(((Fruity)other).fName);
  }
}

public class RQ400A50 {
  public static void main(String[] args) {
    Fruity apple = new Fruity("Apple");
    Fruity orange = new Fruity("Orange");
    List<Fruity> list = new ArrayList<>();
    list.add(apple); list.add(orange); list.add(apple);
    System.out.println("Before: " + list);
    // (1) INSERT CODE HERE
    System.out.println("After:  " + list);
  }
}
```

Which code, when inserted at (1), will result in the following output from the program:

Click here to view code image

```
Before: [Apple, Orange, Apple]
After:  [Orange]
```

Select the two correct answers.

a.

```
for (Fruity f : list) {
  if (f.equals(apple))
    list.remove(f);
}
```

b.

```
int i = 0;
for (Fruity f : list) {
  if (f.equals(apple))

    list.remove(i);
    i++;
  }
```

c. Click here to view code image

```
  for (int j = 0; j < list.size(); j++) {
    Fruity f = list.get(j);
    if (f.equals(apple))
      list.remove(j);
  }
```

d. Click here to view code image

```
  Iterator<Fruity> itr = list.iterator();
  while (itr.hasNext()) {
    Fruity f = itr.next();

    if (f.equals(apple))
      itr.remove();
  }
```

15.3 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
class AnotherListIterator<T> implements Iterable<T>{

  private List<T> lst;
  public AnotherListIterator(List<T> lst) { this.lst = lst; }

  public Iterator<T> iterator() {
    return new Iterator<T>() {
      private int next = lst.size() - 1;

      public boolean hasNext() { return (next >= 0); }
      public T next() {
        T element = lst.get(next);
        next--;
        return element;
      }
    };
  }

  public static void main(String[] args) {
    List<String> lst = List.of("Hi", "Howdy", "Hello");
    AnotherListIterator<String> rlt = new AnotherListIterator<>(lst);
    for (String str : rlt) {
      System.out.print("|" + str + "|");
    }
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw an exception when run.

c. The program will compile and print `|Hi||Howdy||Hello|` at runtime.

d. The program will compile and print `|Hello||Howdy||Hi|` at runtime.

e. The program will compile and print the strings in an unpredictable order at runtime.

15.4 Which of the following statements are true about collections? Select the two correct answers.

a. Methods calling optional operations in a collection must either catch an `UnsupportedOperationException`, or declare it in their `throws` clause.

b. A `List<E>` can have duplicate elements.

c. An `ArrayList<E>` can only accommodate a fixed number of elements.

d. A `Set<E>` allows at most one `null` element.

15.5 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Sets {
  public static void main(String[] args) {
    HashSet<Integer> set1 = new HashSet<>();
    addRange(set1, 1);
    ArrayList<Integer> list1 = new ArrayList<>();
    addRange(list1, 2);
    TreeSet<Integer> set2 = new TreeSet<>();
    addRange(set2, 3);
    ArrayDeque<Integer> deque = new ArrayDeque<>();
    addRange(deque, 5);
    set1.removeAll(list1);
    list1.addAll(set2);
    deque.addAll(list1);
    set1.removeAll(deque);
    System.out.println(set1);
  }
  static void addRange(Collection<Integer> col, int step) {
    for (int i = step * 2; i <= 25; i += step) {
      col.add(i);
    }
  }
}
```

Select the one correct answer.

a. The program will fail to compile, since operations are performed on incompatible collection implementations.

b. The program will fail to compile, since no `Comparator` is supplied to the `TreeSet` constructor for sorting the elements.

c. The program will compile. When run, it will throw an `UnsupportedOperation-Exception`.

d. The program will compile. When run, it will print all primes below 25.

15.6 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Iterate {
  public static void main(String[] args) {

    List<String> l = new ArrayList<>();
    l.add("A"); l.add("B"); l.add("C"); l.add("D"); l.add("E");
    ListIterator<String> i = l.listIterator();
    i.next(); i.next(); i.next(); i.next();
    i.remove();
    i.previous(); i.previous();
    i.remove();
    System.out.println(l);
  }
}
```

Select the one correct answer.

a. It will print `[A, B, C, D, E]`.

b. It will print `[A, C, E]`.

c. It will print `[B, D, E]`.

d. It will print `[A, B, D]`.

e. It will print `[B, C, E]`.

f. It will throw a `NoSuchElementException`.

15.7 Which of the following statements, when inserted independently at (1), will guarantee that the following program will print `[1, 9]`?

Click here to view code image

```
import java.util.*;
public class RightOne {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    collection.add(1); collection.add(9); collection.add(1);
    System.out.println(collection);
  }
}
```

Select the four correct answers.

a. Click here to view code image

```
Collection<Integer> collection = new HashSet<>();
```

b. Click here to view code image

```
Set<Integer> collection = new HashSet<>();
```

c. Click here to view code image

```
HashSet<Integer> collection = new LinkedHashSet<>();
```

d. Click here to view code image

```
Set<Integer> collection = new LinkedHashSet<>();
```

e. Click here to view code image

```
Collection<Integer> collection = new TreeSet<>();
```

f. Click here to view code image

```
NavigableSet<Integer> collection = new TreeSet<>();
```

15.8 Which of the following statements, when inserted independently at (1), will result in program output that does not include the word `"shell"`?

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;
public class RQ400A400 {
  public static void main(String[] args) {
    NavigableSet<String> strSetA = new TreeSet<>();
    Collections.addAll(strSetA, "sea", "shell", "soap", "swan");
    // (1) INSERT STATEMENT HERE
  }
}
```

Select the two correct answers.

a. Click here to view code image

```
out.println(strSetA.headSet("soap", true));
```

b. Click here to view code image

```
out.println(strSetA.headSet("soap", false));
```

c. Click here to view code image

```
out.println(strSetA.tailSet("soap", true));
```

d. Click here to view code image

```
out.println(strSetA.tailSet("soap", false));
```

e. Click here to view code image

```
out.println(strSetA.subSet("sea", false, "soap", true));
```

f. Click here to view code image

```
out.println(strSetA.subSet("sea", true, "soap", false));
```

15.8 Maps

A *map* defines *mappings* from keys to values. The <*key*, *value*> pair is called a *mapping*, also referred to as an *entry*. A map does not allow duplicate keys; in other words, the keys are unique. Each key maps to one value at most, implementing what is called a *single-valued map*. Thus there is a *many-to-one* relationship between keys and values. For example, in a student-grade map, many students (keys) can be awarded the same grade (value), but each student has only one grade. Replacing the value that is associated with a key results in the old entry being removed and a new entry being inserted.

Both the keys and the values must be objects, with primitive values being wrapped in their respective primitive wrapper objects when they are put in a map.

The `Map<K,V>` Interface

A map is not a collection, and the `Map<K, V>` interface does not extend the `Collection<E>` interface. However, the mappings can be viewed as a collection in various ways: a key set, a value collection, or an entry set. A key set view or an entry set view can be iterated over to retrieve the corresponding values from the underlying map (p. 844).

The `Map<K, V>` interface specifies some optional methods. Implementations should throw an `UnsupportedOperationException` if they do not support such an operation. The implementations of maps from the `java.util` package support all the optional operations of the `Map<K, V>` interface (see Table 15.2, p. 788, and Figure 15.3, p. 787).

The `Map<K,V>` interface and its subinterfaces `SortedMap<K,V>` and `NavigableMap<K,V>` provide a versatile set of methods to implement a wide range of operations on maps. Several examples in this section and subsequent sections illustrate many of the methods specified in these interfaces.

Basic Key-Based Operations

These operations constitute the basic functionality provided by a map. Many of the methods will be used in examples presented in this chapter.

> Click here to view code image
>
> ```
> V put(K key, V value)                                    Optional
> default V putIfAbsent(K key, V value)
> ```
>
> The `put()` method inserts the <`key`, `value`> entry into the map. It returns the old *value* previously associated with the specified `key`, if any. Otherwise, it returns the `null` value.
>
> The default method `putIfAbsent()` associates the `key` with

the given `value` and returns `null` if the specified `key` is *not* already associated with a non-`null` value; otherwise, it returns the currently associated value.

Click here to view code image

```
V get(Object key)
default V getOrDefault(Object key, V defaultValue)
```

The `get()` method returns the value to which the specified `key` is mapped, or `null` if no entry is found.

The default method `getOrDefault()` returns the value to which the specified `key` is mapped, or the specified `defaultValue` if this map contains no entry for the `key`.

Click here to view code image

```
V remove(Object key)                               Optional
default boolean remove(Object key, Object value)
```

The `remove()` method deletes the entry for the specified `key`. It returns the *value* previously associated with the specified `key`, if any. Otherwise, it returns the `null` value.

The default method `remove()` removes the entry for the specified `key` and returns `true` only if the specified `key` is currently mapped to the specified `value`.

Click here to view code image

```
default V replace(K key, V value)
default boolean replace(K key, V oldValue, V newValue)
```

In the first `replace()` method, the value associated with the `key` is replaced with the specified `value` only if the `key` is already mapped to some value. It returns the *previous* value associated with the specified `key`, or `null` if there was no entry found for the `key`.

In the second `replace()` method, the value associated with the `key` is replaced with the specified `newValue` only if the `key` is currently mapped to the specified `oldValue`. It returns `true` if the value in the entry for the `key` was replaced with the `newValue`.

Click here to view code image

```
boolean containsKey(Object key)
boolean containsValue(Object value)
```

The `containsKey()` method returns `true` if the specified `key` is mapped to a value in the map.

> The `containsValue()` method returns `true` if there exists one or more keys that are mapped to the specified `value`.

Creating Unmodifiable Maps

Creating unmodifiable maps is analogous to creating unmodifiable lists (§12.2, p. 649) or unmodifiable sets (p. 804). The `Map<K, V>` interface provides factory methods to create *unmodifiable maps* that have the following characteristics:

- Keys and values in such maps cannot be added, removed, or updated. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the keys and values themselves are mutable, the map may appear to be modified.
- The `null` value cannot be used for keys and values, and will result in a `Null-PointerException` if an attempt is made to create such a map with `null` keys or `null` values.
- Duplicate keys are rejected when creating such a map, resulting in an `Illegal-ArgumentException`.
- The iteration order of mappings in such maps is unspecified.
- Such maps are serializable if their keys and values are serializable (§20.5, p. 1261).

> Click here to view code image
>
> ```
> static <K,?V> Map<K,?V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
>                            K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
>                            K k9, V v9, K k10, V v10)
> ```
>
> This method is overloaded, accepting any number of entries (`k, v`) from 0 to 10. It returns an unmodifiable map containing the number of mappings specified. It throws a `NullPointerException` if any key or value is `null`. It throws an `IllegalArgumentException` if there are any duplicate keys.
>
> Click here to view code image
>
> ```
> @SafeVarargs static <K,V> Map<K,V> ofEntries(
>           Map.Entry<? extends K,? extends V>... entries)
> ```
>
> This variable arity method returns an unmodifiable map containing an arbitrary number of entries. It throws a `NullPointerException` if a key or a value is `null`, or if the

variable arity parameter `entries` is `null`. The annotation suppresses the heap pollution warning in its declaration and unchecked generic array creation warning at the call sites. See the method `entry()` below to create individual entries.

Click here to view code image

```
static <K,V> Map.Entry<K,V> entry(K k, V v)
```

This generic method returns an unmodifiable `Map.Entry` object containing the specified key and value. Attempts to create an entry using a `null` key or a `null` value result in a `NullPointerException`.

Each entry—that is, <*key*, *value*> pair—is represented by an object implementing the nested `Map.Entry<K, V>` interface. An entry can be manipulated by methods defined in this interface, which are self-explanatory:

Click here to view code image

```
interface Entry<K, V> {    // Nested interface in the Map<K, V> interface.
    K getKey();

    V getValue();
    V setValue(V value);   // Only if the entry is modifiable.
}
```

```
static <K,V> Map<K,V> copyOf(Map<? extends K, ? extends V> map)
```

This generic method returns an unmodifiable map containing copies of the entries in the specified map. The specified map must not be `null`, and it must not contain any `null` keys or values—otherwise, a `NullPointerException` is thrown. If the specified map is subsequently modified, the returned `Map<K,V>` will not reflect such modifications.

The code below shows that a map created by the `Map.of()` method cannot be modified. We cannot change or remove any entry in the map. Note that the `Map.of()` method allows up to 10 entries, and there is no variable arity `Map.of()` method. The map returned is also not an instance of the `HashMap` class.

Click here to view code image

```
Map<Integer, String> jCourses = Map.of(
                     200, "Basic Java",    300, "Intermediate Java",
                     400, "Advanced Java", 500, "Kickass Java");
//  jCourses.put(200, "Java Jive");                 // UnsupportedOperationExcept
//  jCourses.remove(500);                           // UnsupportedOperationExcept
System.out.println(jCourses instanceof HashMap); // false
System.out.println(jCourses);
// {200=Basic Java, 400=Advanced Java, 300=Intermediate Java, 500=Kickass Java
```

The `Map.of()` method does not allow duplicate keys, and keys or values cannot be `null`:

Click here to view code image

```
Map<Integer, String> coursesMap1
        = Map.of(101, "Java 1.1", 101, "Java 17");  // IllegalArgumentExceptio
Map<Integer, String> coursesMap2
        = Map.of(101, "Java 1.1", 101, null);       // NullPointerException
```

The following code creates unmodifiable map entries, where both the key and the value are immutable:

Click here to view code image

```
//  Map.Entry<Integer, String> e0 = Map.entry(100, null); // NullPointerExcept
Map.Entry<Integer, String> e1 = Map.entry(200, "Basic Java");
Map.Entry<Integer, String> e2 = Map.entry(300, "Intermediate Java");
Map.Entry<Integer, String> e3 = Map.entry(400, "Advanced Java");
Map.Entry<Integer, String> e4 = Map.entry(500, "Kickass Java");
```

The variable arity `Map.ofEntries()` method can be used to create an unmodifiable map from an arbitrary number of unmodifiable entries.

Click here to view code image

```
Map<Integer, String> unmodCourseMap = Map.ofEntries(e1, e2, e3, e4);  // Varar
// {300=Intermediate Java, 500=Kickass Java, 200=Basic Java, 400=Advanced Java
//unmodCourseMap.replace(200, "Java Jive");    // UnsupportedOperationExceptio
//unmodCourseMap.remove(500);                  // UnsupportedOperationExceptio
```

The following code creates mutable course names that we will use in the next map.

Click here to view code image

```
StringBuilder mc1 = new StringBuilder("Basic Java");
StringBuilder mc2 = new StringBuilder("Intermediate Java");
StringBuilder mc3 = new StringBuilder("Advanced Java");
StringBuilder mc4 = new StringBuilder("Kickass Java");
```

The following code creates unmodifiable map entries, where the keys are immutable but the values are mutable:

Click here to view code image

```
Map.Entry<Integer, StringBuilder> me1 = Map.entry(200, mc1);
Map.Entry<Integer, StringBuilder> me2 = Map.entry(300, mc2);
Map.Entry<Integer, StringBuilder> me3 = Map.entry(400, mc3);
Map.Entry<Integer, StringBuilder> me4 = Map.entry(500, mc4);
```

We can use the variable arity `Map.ofEntries()` method to create an unmodifiable map, where trying to replace or remove a course results in an `UnsupportedOperation-Exception`:

Click here to view code image

```
Map<Integer, StringBuilder> unmodMapWithMutableCourses
                = Map.ofEntries(me1, me2, me3, me4);          // Varargs
System.out.println(unmodMapWithMutableCourses);
// {200=Basic Java, 500=Kickass Java, 300=Intermediate Java, 400=Advanced Java

// unmodMapWithMutableCourses.replace(200, mc4); // UnsupportedOperationExcept
// unmodMapWithMutableCourses.remove(400);       // UnsupportedOperationExcept
```

However, the mutable values in the map can be modified:

Click here to view code image

```
StringBuilder mutableCourse = unmodMapWithMutableCourses.get(500);
mutableCourse.replace(0, 7, "Smartass");
```

Click here to view code image

```
System.out.println(unmodMapWithMutableCourses);
// {400=Advanced Java, 500=Smartass Java, 200=Basic Java, 300=Intermediate Jav
```

The code below shows how we can make a copy of a map. The `Map.copyOf()` method creates a copy of the map passed as an argument at (1). The map created is unmodifiable analogous to the maps created with the `Map.of()` or `Map.ofEntries()` methods. The code also shows that modifying the original map does *not* reflect in the copy of the map.

Click here to view code image

```
// Original map:
Map<Integer, StringBuilder> courseMap = new HashMap<>();
courseMap.put(200, mc1); courseMap.put(300, mc2);
courseMap.put(400, mc3); courseMap.put(500, mc4);

// Unmodifiable copy of the map:
Map<Integer, StringBuilder> copyCourseMap = Map.copyOf(courseMap); // (1)

// Modify original map:
courseMap.remove(200);
courseMap.remove(400);
System.out.println("Original: " + courseMap);
System.out.println("Copy: " + copyCourseMap);
```

The code above prints the contents of the maps, showing that the copy of the map was not modified:

Click here to view code image

```
Original: {500=Smartass Java, 300=Intermediate Java}
Copy: {300=Intermediate Java, 500=Smartass Java, 200=Basic Java,
       400=Advanced Java}
```

Advanced Key-Based Operations

The following methods for a map take a *function* (implemented by a lambda expression or a method reference) as a parameter to implement various scenarios that manipulate the value associated with a key.

> Click here to view code image
>
> ```
> default V merge(K key, V value,
>                 BiFunction<? super V,? super V,? extends V> remappingFunc)
> ```
>
> If the specified `key` has no entry or is associated with a `null` value, the method associates the `key` with the specified non-`null value`. Otherwise, it associates the `key` with the result of applying the remapping two-arity function to the specified `value` and the currently associated value, or removes the entry for the `key` if the result is `null`.
>
> Note that if the specified `value` is `null`, a `NullPointerException` is thrown at runtime.
>
> Returns the new value associated with the `key`, or `null` if entry for the `key` is removed.
>
> Click here to view code image
>
> ```
> default V compute(K key,
>                   BiFunction<? super K,? super V,? extends V> remappingFunc)
> ```
>
> The remapping two-arity function is applied to the specified `key` and its currently associated value. If the result is non-`null`, it associates the key with the result. Otherwise, it removes any entry for the `key`.
>
> Returns the computed value associated with the `key`, or `null` if the `key` is removed or is not associated with a value.
>
> Click here to view code image
>
> ```
> default V computeIfAbsent(K key,
>                           Function<? super K,? extends V> mappingFunc)
> ```
>
> If the value associated with the specified key is `null` or the key has no entry, this method applies the given function on the `key`, and the result is only associated with the `key` if the result is non-`null`.
>
> Returns the current (existing or computed) value associated with the specified `key`, or `null` if the computed value is `null`.
>
> Click here to view code image

```
default V computeIfPresent(K key,
                           BiFunction<? super K,? super V,? extends V> remappi
```

If the specified key is already associated with a non-`null` value, the method applies the specified remapping two-arity function to the `key` and its currently associated value. The result is only associated with the `key` if it is non-`null`; *otherwise, the key is removed.*

This method returns the current (existing or computed) value associated with the specified `key`, or `null` if the `key` is absent or removed.

Table 15.6 summarizes typical scenarios when using these advanced key-based methods. Given the `value` associated with a key and the `resultValue` returned by the remapping function if it is executed, the action performed and the value returned by each method are shown for each scenario.

Table 15.6 *Summary of Scenarios Using Advanced Key-Based Operations*

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
(givenValue != null)					
[illegible] *No change. Returns* `null`.					
[illegible] `put(key, resultValue)`, *returns* `resultValue`.					
[illegible] *No change. Returns* `null`, *returns* `givenValue`.					
[illegible] `put(key, resultValue)`, *returns* `resultValue`.					
[illegible] *Not mapping. Returns* `null`, *returns* `givenValue`.					
[illegible] `put(key, resultValue)`, *returns* `resultValue`.					

In the code snippets to illustrate each method, an emergency telephone number map, `etnMap`, is loaded with the same entries. This map is a `HashMap<Integer, String>` that is used to map emergency telephone numbers (`Integer`s) to countries (`String`s) where they are used. (*Disclaimer*: There is no guarantee that the information provided below is correct.)

Click here to view code image

```
Map<Integer, String> etnMap = new HashMap<>();
etnMap.put(112, "Norway");
etnMap.put(999, "UK");
etnMap.put(190, null);
etnMap.put(911, null);
// {112=Norway, 999=UK, 190=null, 911=null}
```

In the code below, six method calls are made corresponding to the column for each method in Table 15.6. Each method call shows the value returned by the method as a comment in the call statement.

The `merge()` method executes as follows:

- If the key is associated with the `null` value in the map, as shown at (3) and (4), or has no entry, as shown at (5) and (6), the key is associated with the *non-*`null` *given value*.
- Otherwise, the remapping two-arity function is computed.
  - If the result is `null`, the entry for the key is removed, as shown at (1).
  - If the result is non-`null`, the key is associated with the result, as shown at (2).

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.merge(112, "Mordor", (oVal, value) -> null);       // (1) null, re
etnMap.merge(999, "Mordor", (oVal, value) -> "Uganda");   // (2) Uganda,
etnMap.merge(190, "Mordor", (oVal, value) -> null);       // (3) Mordor,
etnMap.merge(911, "Mordor", (oVal, value) -> "USA");      // (4) Mordor,
etnMap.merge(100, "Mordor", (oVal, value) -> null);       // (5) Mordor,
etnMap.merge(110, "Mordor", (oVal, value) -> "China");    // (6) Mordor,
// After: {100=Mordor, 999=Uganda, 110=Mordor, 190=Mordor, 911=Mordor}
```

The `compute()` method executes as follows:

- The remapping two-arity function is executed.
  - If this result is non-`null`, as at (8), (10), and (12), this result is associated with the key whether or not the key has an entry in the map.
  - If this result is `null` and the key has an entry in the map, the entry is removed, as at (7) and (9).
  - If this result is `null` and the key has no entry in the map, no action is taken, as at (11).

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.compute(112, (key, oVal) -> null);              // (7) null, remo
etnMap.compute(999, (key, oVal) -> "Uganda");          // (8) Uganda, up
etnMap.compute(190, (key, oVal) -> null);              // (9) null, remo
etnMap.compute(911, (key, oVal) -> "USA");             // (10) USA, upda
etnMap.compute(100, (key, oVal) -> null);              // (11) null, no
etnMap.compute(110, (key, oVal) -> "China");           // (12) China, in
// After: {110=China, 999=Uganda, 911=USA}
```

The `computeIfAbsent()` method executes as follows:

- If the key is associated with a `null` value or has no entry in the map, the remap-ping function is executed.

  - If the result is non-`null`, as at (16) and (18), this result is associated with the key.

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfAbsent(112, key -> null);              // (13) Norway,
etnMap.computeIfAbsent(999, key -> "Uganda");          // (14) UK, no c
etnMap.computeIfAbsent(190, key -> null);              // (15) null, no
etnMap.computeIfAbsent(911, key -> "USA");             // (16) USA, upd
etnMap.computeIfAbsent(100, key -> null);              // (17) null, no
etnMap.computeIfAbsent(110, key -> "China");           // (18) China, i
// After: {112=Norway, 110=China, 999=UK, 190=null, 911=USA}
```

The `computeIfPresent()` method executes as follows:

- If the key is associated with a non-`null` value in the map, as at (19) and (20), the remapping two-arity function is executed.
  - If the result is `null`, then the entry with the key is removed, as at (19).
  - If the result is non-`null`, then the key is associated with the result, as at (20).

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfPresent(112, (key, oVal) -> null);      // (19) null, removed
etnMap.computeIfPresent(999, (key, oVal) -> "Uganda");  // (20) Uganda, update
etnMap.computeIfPresent(190, (key, oVal) -> null);      // (21) null, no chang
etnMap.computeIfPresent(911, (key, oVal) -> "USA");     // (22) null, no chang
etnMap.computeIfPresent(100, (key, oVal) -> null);      // (23) null, no actio
etnMap.computeIfPresent(110, (key, oVal) -> "China");   // (24) null, no actio
// After: {999=Uganda, 190=null, 911=null}
```

Bulk Operations

Bulk operations can be performed on an entire map.

```
int size()
boolean isEmpty()
```

Return the number of entries (i.e., number of unique keys in the map) and whether the map is empty or not, respectively.

Click here to view code image

```
void clear()                                     Optional
void putAll(Map<? extends K, ? extends V> map)   Optional
```

The first method deletes all entries from the current map.

The second method copies all entries from the specified `map` to the current map. If a key from the specified map is already in the current map, its associated value in the current map is replaced with the associated value from the specified map.

Click here to view code image

```
default void forEach(BiConsumer<? super K,? super V> action)
```

The specified `action` is performed for each entry in this map until all entries have been processed or the `action` throws an exception. Iteration is usually performed in entry set iteration order. The functional interface `BiConsumer<T,U>` is covered in §13.7, p. 711, where this method is illustrated. It is also used in Example 15.10, p. 844.

Click here to view code image

```
default void replaceAll(BiFunction<? super K,? super V,? extends V> func)
```

The value of each entry is replaced with the result of invoking the given twoarity function on the entry until all entries have been processed or the function throws an exception. The functional interface `BiFunction<T,U,R>` is covered in §13.9, p. 717, where this method is illustrated.

Collection Views

Views allow information in a map to be represented as collections. Elements can be removed from a map via a view, but cannot be added. An iterator over a view will throw an exception if the underlying map is modified concurrently. Example 15.10, p. 844, illustrates collection views.

Click here to view code image

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

Create different views of a map. Changes in the map are reflected in the view, and vice versa. These methods return a set view of keys, a collection view of values, and a set view of <*key*, *value*> entries, respectively. Note that the `Collection` returned by the `values()` method is not a `Set`, as several keys can map to the same value—that is, duplicate values can be included in the returned collection. An entry in the entry set

> view can be manipulated by the methods defined in the `Map.Entry<K, V>` interface.
>
> Example 15.10, p. 844, provides an example of using collection views.

15.9 Map Implementations

Figure 15.3, p. 787, shows four implementations of the `Map<K, V>` interface found in the `java.util` package: `HashMap<K, V>`, `LinkedHashMap<K, V>`, `Hashtable<K, V>`, and `TreeMap<K, V>`.

The classes `HashMap<K, V>` and `Hashtable<K, V>` implement unordered maps. The class `LinkedHashMap<K, V>` implements ordered maps, and the class `TreeMap<K, V>` implements sorted maps (p. 845).

While the `HashMap<K, V>` class is not thread-safe and permits a `null` key and `null` values (analogous to the `LinkedHashMap<K, V>` class), the `Hashtable<K, V>` class is thread-safe and permits only non-`null` keys and values (see Table 15.7). The thread-safety provided by the `Hashtable<K, V>` class comes with a performance penalty. Thread-safe use of maps is also provided by the methods in the `Collections` class (p. 856). Like the `Vector<E>` class, the `Hashtable<K, V>` class is also a legacy class that has been retrofitted to implement the `Map<K, V>` interface.

Table 15.7 *Map Implementations*

Map	**Kind of map**	**Thread-safe?**	**null key?**	**null values?**
`HashMap<K,V>`	*Unordered*	*Not thread-safe*	*Allowed*	*Allowed*
`LinkedHashMap<K,V>` `extends HashMap<K,V>`	*Key insertion order/Access order*	*Not thread-safe*	*Allowed*	*Allowed*
`Hashtable<K,V>`	*Unordered*	*Thread-safe*	*Not allowed*	*Not allowed*
`TreeMap<K,V>`	*Key sort order*	*Not thread-safe*	*Not allowed*	*Allowed*

These map implementations are based on a hashing algorithm. Operations on a map thus rely on the `hashCode()` and `equals()` methods of the key objects (§14.2, p. 744, and §14.3, p. 753).

The `LinkedHashMap<K, V>` implementation is a subclass of the `HashMap<K, V>` class. The relationship between the map classes `LinkedHashMap<K, V>` and `HashMap<K, V>` is analogous to the relationship between their counterpart set classes `LinkedHashSet<E>` and `HashSet<E>`. The entries of a `HashMap<K, V>` (analogous to a `HashSet<E>`) are unordered. The entries of a `LinkedHashMap<K, V>` (analogous to a `LinkedHashSet<E>`) are ordered. By default, the entries of a `LinkedHashMap<K, V>` are in *key insertion order*— that is, the order in which the keys are inserted in the map. This order does not change if a key is reinserted because no new entry is created if the key's entry already exists. The elements in a `LinkedHashSet<E>` are also at (element) insertion order. However, a `LinkedHashMap<K, V>` can also maintain its entries in *access order*— that is, the order in which its entries are accessed, from least-recently accessed to most-recently accessed entries. This *ordering mode* can be specified in one of the constructors of the `LinkedHashMap<K, V>` class.

Both the `HashMap<K, V>` and the `LinkedHashMap<K, V>` classes provide comparable performance, but the `HashMap<K, V>` class is the natural choice if ordering is not an issue. Operations such as adding, removing, or finding an entry based on a key are in constant time, as these are based on hashing the key. Operations such as finding the entry with a specific *value* are in linear time, as these involve searching through the entries.

Adding, removing, and finding entries in a `LinkedHashMap<K, V>` can be slightly slower than in a `HashMap<K, V>`, as an ordered doubly linked list has to be maintained. Iteration over a map is through one of its collection views. For an underlying `LinkedHashMap<K, V>`, the iteration time is proportional to the size of the map— regardless of its capacity. However, for an underlying `HashMap<K, V>`, it is proportional to the capacity of the map.

The concrete map implementations override the `toString()` method. The standard text representation generated by the `toString()` method for a map is

$$\{key_1=value_1,\ key_2=value_2,\ \ldots,\ key_n=value_n\}$$

where each key_i and each $value_i$, where `1 <= i <= n`, is the text representation generated by the `toString()` method of the individual key and value objects in the map, respectively.

As was the case with collections, implementation classes provide a standard constructor that creates a new empty map, and a constructor that creates a new map based on an existing map. Additional constructors create empty maps with given initial capacities and load factors. The `HashMap<K, V>` class provides the following constructors:

Click here to view code image

```
HashMap()
HashMap(int initialCapacity)
```

> ```
> HashMap(int initialCapacity, float loadFactor)
> ```
>
> Constructs an empty `HashMap`, using either a default or specified initial capacity and a load factor.
>
> Click here to view code image
>
> ```
> HashMap(Map<? extends K,? extends V> otherMap)
> ```
>
> Constructs a new map containing the elements in the specified map.

The `LinkedHashMap<K, V>` and `Hashtable<K, V>` classes have constructors analogous to the four constructors for the `HashMap<K, V>` class. In addition, the `LinkedHashMap<K, V>` class provides a constructor where the ordering mode can also be specified:

> Click here to view code image
>
> ```
> LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
> ```
>
> Constructs a new, empty `LinkedHashMap` with the specified initial capacity, the specified load factor, and the specified ordering mode. The ordering mode is `true` for *access order* and `false` for *key insertion order*.

Example 15.10 prints a textual histogram for the frequency of weight measurements in a weight group, where a weight group is defined as an interval of five units. The weight measurements are supplied as program arguments. A `HashMap<Integer, Integer>` is used, where the key is the weight group and the value is the frequency. The example illustrates many key-based operations on maps, the creation of key views, and iteration over a map.

We have intentionally used a `HashMap<K,V>`. It is instructive to compare this with the solution in Example 15.11 that uses a `TreeMap<K,V>` that simplifies the solution further, when the entries are maintained in key-sort order.

The program proceeds as follows:

- An empty `HashMap<Integer, Integer>` is created at (1), where the key is the weight group and the associated value is the frequency.
- A `for(:)` loop is used at (2) to read the weights specified as program arguments, converting each weight to its corresponding weight group and updating the frequency of the weight group.

Each program argument is parsed to a `double` value at (3), which is then used to determine the correct weight group at (4). The call to the `merge()` method at (5) updates the frequency map:

Click here to view code image

```
// With method reference:
groupFreqMap.merge(weightGroup, 1, Integer::sum);                          //
// With lambda expression:
groupFreqMap.merge(weightGroup, 1,
   (oldVal, givenVal) -> Integer.sum(oldVal, givenVal));
```

If the weight group is not in the map, its frequency is set to `1`; otherwise, the method reference `Integer::sum` increments the frequency by the given value `1`. In both cases an appropriate entry for the weight group is put in the frequency map. Note the arguments passed to the method reference and the lambda expression: `oldVal` is the current value associated with the key and `givenVal` is the specified value, `1`, passed to the `merge()` method.

Generic types guarantee that the keys and the values in the map are of the correct type, and autoboxing/unboxing of primitive values guarantees the correct type of an operand in an expression.

Some other strategies to update the frequency map are outlined here, but none is more elegant than the one with the `merge()` method.

The straightforward solution below requires an explicit `null` check to determine whether the value returned by the `get()` method is `null`, or risks a `NullPointer-Exception` at runtime when incrementing the `null` value in the `frequency` reference.

Click here to view code image

```
Integer frequency = groupFreqMap.get(weightGroup);
if (frequency == null) frequency = 0;
groupFreqMap.put(weightGroup, ++frequency);
```

Below, the `getOrDefault()` method never returns a `null` value, since it returns the associated frequency value or the specified default value `0` depending on whether the weight group is found or not found in the map, respectively. No explicit `null` check is necessary.

Click here to view code image

```
Integer frequency = groupFreqMap.getOrDefault(weightGroup, 0);
groupFreqMap.put(weightGroup, ++frequency);
```

The `putIfAbsent()` method puts an entry with frequency `0` for the weight group if it is not found so that the `get()` method will always return a non-`null` value which can be safely incremented.

Click here to view code image

```
groupFreqMap.putIfAbsent(weightGroup, 0);
Integer frequency = groupFreqMap.get(weightGroup);
groupFreqMap.put(weightGroup, ++frequency);
```

The `compute()` method always updates the value associated with the key if the two-arity function returns a non-`null` value, which is always the case below. The `null` check is now done in the lambda expression.

Click here to view code image

```
groupFreqMap.compute(weightGroup, (k, v) -> v == null ? 1 : v + 1);
```

- The program creates a sorted set of keys (which are weight groups) from the `groupFreqMap` at (6). The `Map.keySet()` method returns a set view of keys, which is passed as an argument to a `TreeSet<E>` to create a sorted set—in this case, the natural ordering of `Integer`s is used.

Click here to view code image

```
TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet()); /
```

- The histogram is printed by implicitly iterating over the sorted key set at (7). Since a sorted set is an `Iterable<E>`, we can use the `forEach()` method that takes a consumer to print the histogram.

Click here to view code image

```
sortedKeySet.forEach(key ->
    System.out.printf("%5s: %s%n", key,
        String.join("", Collections.nCopies(groupFreqMap.get(key), "*")
```

For each key, the corresponding value (i.e., the frequency) is retrieved and converted to a string with the corresponding number of `"*"`. The method `Collections.nCopies()` creates a list with an equal number of elements as its first argument and where each element is the same as the second argument. The elements of this list are joined to create a string by the `String.join()` method with the first argument specifying the delimiter to use between the elements, which in this case is the empty string.

Alternately, a `for(:)` loop can be used to explicitly iterate over the sorted set view of the frequency map.

Example 15.10 *Using Maps*

Click here to view code image

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeSet;

public class Histogram {
  public static void main(String[] args) {
    System.out.println("Data: " + Arrays.toString(args));

    // Create a map to store the frequency for each group.
    Map<Integer, Integer> groupFreqMap = new HashMap<>();                  //

    // Determine the frequencies:
    for (String argument : args) {                                         //
      double weight = Double.parseDouble(argument);                        //
      int weightGroup = (int) Math.round(weight/5.0)*5;                    //
      groupFreqMap.merge(weightGroup, 1, Integer::sum);                    //
    }
    System.out.println("Frequencies: " + groupFreqMap);

    // Create sorted key set.
    TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet());  //

    System.out.println("Histogram:");
    // Implicit iteration over the key set.
    sortedKeySet.forEach(key ->                                            //
        System.out.printf("%5s: %s%n", key,
            String.join("", Collections.nCopies(groupFreqMap.get(key), "*")))
    );
  }
}
```

Running the program with the following arguments:

Click here to view code image

```
>java Histogram 74 75 93 75 93 82 61 92 10 185
```

gives the following output:

Click here to view code image

```
Data: [74, 75, 93, 75, 93, 82, 61, 92, 10, 185]
Frequencies: {80=1, 185=1, 10=1, 90=1, 75=3, 60=1, 95=2}
Histogram:
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
```

15.10 Sorted Maps and Navigable Maps

The `SortedMap<K, V>` interface extends the `Map<K, V>` interface, and the `NavigableMap<K, V>` interface extends the `SortedMap<K, V>` interface. The two maps are analogs of the `SortedSet<E>` and the `NavigableSet<E>` interfaces, respectively.

The `SortedMap<K, V>` Interface

The `SortedMap<K, V>` interface extends the `Map<K, V>` interface to provide the functionality for implementing maps with *sorted keys*. Its operations are analogous to those of the `SortedSet<E>` interface (p. 810), applied to maps and keys rather than to sets and elements.

> Click here to view code image
>
> ```
> // First-last keys
> K firstKey()                                      Sorted set: first()
> K lastKey()                                       Sorted set: last()
> ```
>
> Return the first (smallest) key and the last (largest) key in the sorted map, respectively, dependent on the key-sort order in the map. They throw a `NoSuchElementException` if the map is empty.
>
> Click here to view code image
>
> ```
> // Range-view operations
> SortedMap<K,V> headMap(K toKey)                   Sorted set: headSet()
> SortedMap<K,V> tailMap(K fromKey)                 Sorted set: tailSet()
> SortedMap<K,V> subMap(K fromKey, K toKey)         Sorted set: subSet()
> ```
>
> Return different views analogous to those for a `SortedSet<E>`. The views returned are a portion of the map whose keys are strictly less than `toKey`, greater than or equal to `fromKey`, and between `fromKey` (inclusive) and `toKey` (exclusive), respectively. That is, these partial map views include `fromKey` if it is present in the map, but `toKey` is excluded.
>
> Click here to view code image
>
> ```
> // Comparator access
> Comparator<? super K> comparator()
> ```
>
> Returns the key comparator that defines the key-sort order, or `null` if the sorted map uses natural ordering for the keys.

The `NavigableMap<K, V>` Interface

Analogous to the `NavigableSet<E>` interface extending the `SortedSet<E>` interface, the `NavigableMap<K, V>` interface extends the `SortedMap<K, V>` interface with navigation methods to find the closest matches for specific search targets. The `NavigableMap<K, V>` interface replaces the `SortedMap<K, V>` interface and is the preferred choice when a sorted map is needed.

In addition to the methods of the `SortedMap<K, V>` interface, the `NavigableMap<K, V>` interface adds the *new* methods shown below, where the analogous methods from the `NavigableSet<E>` interface are also identified. Note that where a `NavigableMap<K, V>` method returns a `Map.Entry` object representing an entry, the corresponding `NavigableSet<E>` method returns an element of the set.

> Click here to view code image
>
> ```
> // First-last entries
> // Remove
> Map.Entry<K, V> pollFirstEntry()                   Navigable set:  pollFirst()
> Map.Entry<K, V> pollLastEntry()                    Navigable set:  pollLast()
>
> // Examine
> Map.Entry<K, V> firstEntry()
> Map.Entry<K, V> lastEntry()
> ```
>
> The `pollFirstEntry()` method *removes* and returns the first *entry*, and the `poll-LastEntry()` method removes and returns the last *entry* currently in this navigable map. The entry is determined according to the ordering policy employed by the map—for example, natural ordering. Both return `null` if the navigable set is empty. The last two methods only retrieve, and do not remove, the value that is returned.
>
> Click here to view code image
>
> ```
> // Range-view operations
> NavigableMap<K, V> headMap(K toKey,                   Navigable set:  headSet()
>                      boolean inclusive)
> NavigableMap<K, V> tailMap(K fromKey,                 Navigable set:  tailSet()
>                      boolean inclusive)
> NavigableMap<K, V> subMap(K fromKey,                  Navigable set:  subSet()
>                      boolean fromInclusive,
>                      K toKey,
>                      boolean toInclusive)
> ```
>
> These operations are analogous to the ones in the `SortedMap<K, V>` interface (p. 845), returning different views of the underlying navigable map, depending on the bound elements. However, the bound elements can be *excluded or included* by the operation, depending on the value of the `boolean` argument `inclusive`.

Click here to view code image

```
// Closest-matches
Map.Entry<K, V> ceilingEntry(K key)                    Navigable set: ceiling()
K               ceilingKey(K key)
Map.Entry<K, V> floorEntry(K key)                      Navigable set: floor()
K               floorKey(K key)
Map.Entry<K, V> higherEntry(K key)                     Navigable set: higher()
K               higherKey(K key)
Map.Entry<K, V> lowerEntry(K key)                      Navigable set: lower()
K               lowerKey(K key)
```

The ceiling methods return the least entry (or key) in the navigable map >= to the argument `key`. The floor methods return the greatest entry (or key) in the navigable map <= to the argument `key`. The higher methods return the least entry (or key) in the navigable map > the argument `key`. The lower methods return the greatest entry (or key) in the navigable map < the argument `key`. All methods return `null` if there is no such `key`.

Click here to view code image

```
// Navigation-views
NavigableMap<K, V> descendingMap()                 Navigable set: descendingSet()
NavigableSet<K> descendingKeySet()
NavigableSet<K> navigableKeySet()
```

The first method returns a *reverse-order map view* of the entries in the navigable map. The second method returns a *reverse-order key set view* for the entries in the navigable map. The last method returns an *ascending-order key set view* for the entries in the navigable map.

The `TreeMap<K,V>` Class

The `TreeMap<K, V>` class is the analog of the `TreeSet<E>` class (p. 812), but in this case for maps. It provides an implementation that sorts its entries in a specific order (see also Figures 15.2 and 15.3, p. 786).

The `TreeMap<K, V>` class implements the `NavigableMap<K, V>` interface, and thereby the `SortedMap<K, V>` interface. By default, operations on sorted maps rely on the natural ordering of the keys. However, a total ordering can be specified by passing a customized comparator to the constructor.

A key in a `TreeMap<K, V>` cannot have the `null` value, but the value associated with a key in an entry can be `null`.

The `TreeMap<K, V>` implementation uses balanced trees, which deliver excellent performance for all operations. However, searching in a `HashMap<K, V>` can be faster than in a `TreeMap<K, V>`, as hashing algorithms usually offer better performance than the search algorithms for balanced trees.

The `TreeMap<K, V>` class provides four constructors, analogous to the ones in the `TreeSet<E>` class:

```
TreeMap()
```

A standard constructor used to create a new empty sorted map, according to the natural ordering of the keys.

Click here to view code image

```
TreeMap(Comparator<? super K> c)
```

A constructor that takes an explicit comparator for the keys, that is used to order the entries in the map.

Click here to view code image

```
TreeMap(Map<? extends K, ? extends V> map)
```

A constructor that can create a sorted map based on a map, according to the natural ordering of the keys of the specified map.

Click here to view code image

```
TreeMap(SortedMap<K, ? extends V> map)
```

A constructor that creates a new map containing the same entries as the specified sorted map, with the same ordering for the keys as the specified map.

Example 15.11 illustrates using navigable maps. It also prints a textual histogram like the one in Example 15.10, but using a `TreeMap<K, V>`, and in addition, it prints some statistics about the navigable map. See also the output from running the example. Some remarks about Example 15.11:

- An empty `NavigableMap<Integer, Integer>` is created at (1), where the key is the weight group and the value is the frequency. The loop at (2) reads the weights specified as program arguments, and creates a frequency map, as described in Example 15.10.
- The method `printHistogram()` at (15) prints a histogram of the frequencies in a navigable map:

Click here to view code image

```
public static <K> void printHistogram(NavigableMap<K, Integer> freqMap)
```

It is a generic method with one type parameter, `K`, that specifies the type of the keys, and the type of the values (i.e., frequencies) is `Integer`. It prints the map and the number of entries in the map at (16) and (17), respectively. The `forEach()` method is invoked on the map at (18) to iterate over the *entries* in order to print the histogram, as explained in Example 15.10.

Printing the histogram at (3) shows the number of entries ordered in ascending key order and the size of the map:

Click here to view code image

```
Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
No. of weight groups: 7
...
```

- Calls to the methods `firstEntry()` and `lastEntry()` at (4) and (5):

Click here to view code image

```
out.println("First entry: " + groupFreqMap.firstEntry());              /
out.println("Last entry: " + groupFreqMap.lastEntry());                /
```

return the following entries, respectively:

```
First entry: 10=1
Last entry: 185=1
```

- Calls to the methods `floorEntry()` and `higherKey()` with the key value `77` at (6) and with the key value `90` at (7), respectively:

Click here to view code image

```
out.println("Greatest entry <= 77: "
           + groupFreqMap.floorEntry(77));                             /
out.println("Smallest key > 90: "
           + groupFreqMap.higherKey(90));                              /
```

give the following output, respectively:

Click here to view code image

```
Greatest entry <= 77: 75=3
Smallest key > 90: 95
```

- Calls to the methods `tailMap(75, true)` and `headMap(75, false)` at (8) and (9) with the argument `75`:

Click here to view code image

```
...
```

```
printHistogram(groupFreqMap.tailMap(75, true));
...
printHistogram(groupFreqMap.headMap(75, false));
```

return the following map views, respectively:

Click here to view code image

```
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
...
Head map (Groups <  75): {10=1, 60=1}
...
```

- The call to the `subMap()` method at (10)

  Click here to view code image

  ```
  printHistogram(groupFreqMap.subMap(
       groupFreqMap.firstEntry().getKey(), false,
       groupFreqMap.lastEntry().getKey(), false));
  ```

  returns the map view with the first and the last entry excluded:

  Click here to view code image

  ```
  Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1,
  ...
  ```

- Polling of a navigable map is shown at (11). Polling is done directly on the navigable map, and the first entry in the map is always retrieved and removed by the `pollFirstEntry()` method at (12). A `while` loop is used to iterate over the entries in the map until the map is empty. For each entry, its key and its value is printed.

  Click here to view code image

  ```
  // Poll the navigable map:
  int sumValues = 0;
  while (!groupFreqMap.isEmpty()) {
    Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();

    Integer frequency = entry.getValue();
    sumValues += frequency;
    out.printf("%5s: %s%n", entry.getKey(), frequency);
  }
  ```

  The number of weights—that is, the sum of the values in the map—is also calculated in the loop and printed at (13) and (14), respectively. After polling the map, the output shows that the map is empty.

Example 15.11 *Using Navigable Maps*

Click here to view code image

```
import java.util.Collections;
```

```
import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;
import static java.lang.System.out;

public class HistogramStats {
  public static void main(String[] args) {

    // Create a navigable map to store the frequency for each group.
    NavigableMap<Integer, Integer> groupFreqMap = new TreeMap<>();        //

    // Determine the frequencies:
    for (String argument : args) {                                         //
      double weight = Double.parseDouble(argument);
      int weightGroup = (int) Math.round(weight/5.0)*5;
      groupFreqMap.merge(weightGroup, 1, Integer::sum);
    }

    // Print statistics about the frequency map:
    out.print("Group frequency map: ");
    printHistogram(groupFreqMap);                                          //

    out.println("First entry: " + groupFreqMap.firstEntry());              //
    out.println("Last entry: " + groupFreqMap.lastEntry());                //
    out.println("Greatest entry <= 77: "
                + groupFreqMap.floorEntry(77));                            //
    out.println("Smallest key > 90: "
                + groupFreqMap.higherKey(90));                             //

    out.print("Tail map (Groups >= 75): ");
    printHistogram(groupFreqMap.tailMap(75, true));                        //

    out.print("Head map (Groups <  75): ");
    printHistogram(groupFreqMap.headMap(75, false));                       //

    out.print("Frequency map (first and last entry excluded): ");
    printHistogram(groupFreqMap.subMap(                                    //
        groupFreqMap.firstEntry().getKey(), false,
        groupFreqMap.lastEntry().getKey(), false));

    // Poll the navigable map:
    out.println("Histogram (by polling):");
    int sumValues = 0;
    while (!groupFreqMap.isEmpty()) {
      Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();   //
      Integer frequency = entry.getValue();
      sumValues += frequency;
      out.printf("%5s: %s%n", entry.getKey(), frequency);
    }
    out.println("Number of weights registered: " + sumValues);             //
    out.println("Group frequency map after polling: " + groupFreqMap);     //
  }

  // Prints a histogram for entries in a navigable map.
  public static <K> void printHistogram(NavigableMap<K, Integer> freqMap) {//
    out.println(freqMap);                                                  //
    out.println("No. of entries: " + freqMap.size());                      //
    freqMap.forEach((k, v) ->                                              //
        out.printf("%5s: %s%n", k,
                   String.join("", Collections.nCopies(v, "*"))));
  }
}
```

Running the program with the following arguments:

Click here to view code image

```
>java HistogramStats 74 75 93 75 93 82 61 92 10 185
```

gives the following output:

Click here to view code image

```
Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
No. of entries: 7
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
First entry: 10=1
Last entry: 185=1
Greatest entry <= 77: 75=3
Smallest key > 90: 95
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
No. of entries: 5
   75: ***

   80: *
   90: *
   95: **
  185: *
Head map (Groups <  75): {10=1, 60=1}
No. of entries: 2
   10: *

   60: *
Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1, 95=2}
No. of entries: 5
   60: *

   75: ***
   80: *
   90: *
   95: **
Histogram (by polling):
   10: 1

   60: 1
   75: 3
   80: 1
   90: 1
   95: 2
  185: 1
Number of weights registered: 10
Group frequency map after polling: {}
```

Review Questions

15.9 Which of the following statements are true about maps?

Select the two correct answers.

a. The return type of the `values()` method is `Set<V>`.

b. Changes made in the set view returned by the `keySet()` method will be reflected in the underlying map.

c. The `Map<K, V>` interface extends the `Collection<E>` interface.

d. All keys in a map are unique.

e. All `Map<K, V>` implementations maintain the keys in some sort order.

15.10 Which of the following methods are defined by the `java.util.Map.Entry<K, V>` interface?

Select the three correct answers.

a. `K getKey()`

b. `K setKey(K value)`

c. `V getValue()`

d. `V setValue(V value)`

e. `void set(K key, V value)`

15.11 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump1 {
  public static void main(String[] args) {

    NavigableSet<String> set = new TreeSet<>(Collections.reverseOrder())
    Collections.addAll(set, "Step", "Jump", "Step", "Hop");
    System.out.println(set);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Step, Step, Jump, Hop]`

b. `[Step, Jump, Hop]`

c. `[Hop, Jump, Step, Step]`

d. `[Hop, Jump, Step]`

e. The result is unpredictable.

f. The code will throw an exception at runtime.

15.12 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump2 {
  public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder(
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>(set1);
    System.out.println(set2);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Hop, Jump, Step]`

b. `[Step, Jump, Hop]`

c. The program will compile, but it will produce an unpredictable result when run.

d. The program will throw an exception at runtime.

e. The program will fail to compile.

15.13 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump3 {
  public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder()
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
    System.out.println(set2);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Hop, Jump, Step]`

b. `[Step, Jump, Hop]`

c. The program will compile, but it will produce an unpredictable result when run.

d. The program will throw an exception at runtime.

e. The program will fail to compile.

15.14 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump4 {
    public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder()
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
    while (!set1.isEmpty()) {
      System.out.print(set1.pollLast() + " ");
    }
    while (!set2.isEmpty()) {
      System.out.print(set2.pollFirst() + " ");
    }
  }
}
```

What is the result?

Select the one correct answer.

a. `Hop Jump Step Hop Jump Step`
b. `Hop Jump Step Step Hop Jump`
c. `Step Jump Hop Step Jump Hop`
d. `Step Jump Hop Hop Jump Step`
e. The program will throw an exception at runtime.

15.15 What will be the output of the following program when compiled and run?

Click here to view code image

```
import java.util.*;
public class MapModify {
  public static void main(String[] args) {

    NavigableMap<String, Integer> grades = new TreeMap<>();
    grades.put("A",  5); grades.put("B", 10); grades.put("C", 15);
    grades.put("D", 20); grades.put("E", 25);

    System.out.print(grades.get(grades.firstKey()) + " ");
    System.out.print(sumValues(grades.headMap("D")) + " ");
    System.out.print(sumValues(grades.subMap("B", false, "D", true)) + "
    grades.subMap(grades.firstKey(), false, grades.lastKey(), false).cle
    System.out.println(sumValues(grades));
  }
  public static <K, M extends Map<K, Integer>> int sumValues(M freqMap)
    return freqMap.values().stream().mapToInt(i->i).sum();
  }
}
```

Select the one correct answer.

a. `5 50 35 30`
b. `5 30 35 30`
c. `5 30 25 30`
d. `5 30 35 75`

15.16 Which code, when inserted independently at (1), will result in the following output from the program: `{Soap=10, Salts=30}`?

Click here to view code image

```
import java.util.*;
public class Mapping {
  public static void main(String[] args) {

    NavigableMap<String, Integer> myMap
                  = new TreeMap<>(Collections.reverseOrder());
    myMap.put("Soap", 10); myMap.put("Shampoo", 20); myMap.put("Salts",
    // (1) INSERT CODE HERE
    System.out.println(myMap);
  }
}
```

Select the three correct answers.

a. Click here to view code image

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
    myMap.remove("Shampoo");
```

b. Click here to view code image

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();)
  if (iterator.next().equals("Shampoo"))
    iterator.remove();
```

c. Click here to view code image

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();) {
  if (iterator.next().equals("Shampoo"))
    myMap.remove("Shampoo");
```

d. Click here to view code image

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
    myMap.remove(entry);
```

e. Click here to view code image

```
myMap.subMap("Shampoo", true, "Shampoo", true).clear();
```

f. Click here to view code image

```
myMap.compute("Shampoo", (k, v) -> null);
```

15.17 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Valuables {
  public static void main(String[] args) {

    NavigableMap<Integer, Integer> iMap = new TreeMap<>();
    iMap.put(100, 1); iMap.put(200, 2); iMap.put(300, 3);
    int sumVal = 0;
    iMap.forEach((k, v) -> sumVal += v);
    System.out.println(sumVal);
  }
}
```

Select the one correct answer.

a. The program will compile and print `6` when run.

b. The program will fail to compile because the `forEach()` method requires a `Function<T, R>` and not a `BiFunction<T, U, R>`.

c. There is no method named `forEach` for maps, only for collections.

d. The program will fail to compile because the syntax of the lambda expression in the argument to the `forEach()`

method is not correct.

e. None of the above

15.18 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class InitMap {
  public static void main(String[] args) {

    Map<Integer, NavigableSet<String>> etnMap = new TreeMap<>();
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("USA");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Canada");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Argentina")
    etnMap.computeIfAbsent(112, key -> new TreeSet<>()).add("Norway");
    System.out.println(etnMap);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will result in a runtime exception from one of the calls to the `computeIfAbsent()` method.

c. The program will compile and print the following when run: {112 = [Norway], 911 = [Argentina, Canada, USA]}

d. The program will compile and print the following when run: {112 = [Norway], 911 = [USA]}

e. The program will compile and print the following when run: `{112=Norway, 911=USA}`

f. None of the above

15.19 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ13 {
  public static void main(String[] args) {

    Map<Integer, String> values = new HashMap<>();
    values.put(1, "ONE");   values.put(2, "TWO");
    values.put(3, "THREE"); values.put(4, "FOUR");
    values.replaceAll((k, v) -> {
      switch (k) {
        case 1: return "FIRST";
        case 2: return "SECOND";
        case 3: return "THIRD";
        case 4: return "FOURTH";
      }
      return "ZERO";
    });
    values.forEach((Integer x, String y) -> { System.out.print(y + " ");
  }
}
```

What is the result?

Select the one correct answer.

a. `ONE TWO THREE FOUR`
b. `FIRST SECOND THIRD FOURTH`
c. `1 2 3 4`
d. The program will compile, but it will produce no result when run.
e. The program will throw an exception at runtime.
f. The program will fail to compile.

15.11 The `Collections` Class

The Java Collections Framework also contains two utility classes, `Collections` and `Arrays`, that provide various operations on collections and arrays, such as algorithms for sorting and searching, or creating customized collections. Practically any operation on a collection can be done using the methods provided by this framework.

The methods also throw a `NullPointerException` if the specified collection or array references passed to them are `null`.

The utility methods declared in the `Collections` *class are all* `public` *and* `static`; *therefore, these two modifiers will be omitted in their method header declarations in this section.*

Unmodifiable Views of Collections

Unmodifiable views of collections are created by the utility methods `unmodifiable`*Interface*`()` in the `Collections` class, where *Interface* can be any of the core interfaces in the Java Collections Framework. The name of these methods is a slight misnomer, as they create unmodified *views* of collections on the *underlying* or *backing collection* that is passed as an argument to the method.

Click here to view code image

```
<E> Collection<E>   unmodifiableCollection(Collection<? extends E> c)
<E> List<E>         unmodifiableList(List<? extends E> list)
<E> Set<E>          unmodifiableSet(Set<? extends E> set)
<E> SortedSet<E>    unmodifiableSortedSet(SortedSet<E> sortedSet)
<E> NavigableSet<E> unmodifiableNavigableSet(NavigableSet<E> navSet)
```

Return an *unmodifiable view* of the collection passed as an argument.

Query operations on the returned collection are delegated to

> the *underlying collection*, and changes in the underlying collection are reflected in the unmodifiable view.
>
> Any attempt to modify the view of a collection will result in an `Unsupported-OperationException`.
>
> The view collection returned by the `unmodifiableCollection()` method does not delegate the `equals()` and the `hashCode()` methods to the backing collection. Instead, the returned view uses the corresponding methods inherited from the `Object` class. This is to safeguard the contract of these methods when the backing collection is a set or a list. However, the views returned by the other methods above do not exhibit this behavior.
>
> Click here to view code image
>
> ```
> <K, V> Map<K, V>        unmodifiableMap(Map<? extends K, ? extends V> map)
> <K, V> SortedMap<K, V>  unmodifiableSortedMap(SortedMap<K, ? extends V> sm)
> <K, V> NavigableMap<K, V>
>                   unmodifiableNavigableMap(NavigableMap<K, ? extends V> nm)
> ```
>
> Return an *unmodifiable view* of the map passed as an argument.
>
> Query operations on the view of the map are delegated to the *underlying map*, and changes in the underlying map are reflected in the unmodifiable view.
>
> Any attempt to modify the returned map will result in an `Unsupported-OperationException`.

An *unmodifiable view of a collection* should not be confused with an *unmodifiable collection*. They are both unmodifiable—that is, they are *read-only* collections. No operations that can change them *structurally* are permitted (the famous `UnsupportedOperationException`). However, an unmodifiable view of a collection is *backed* by an *underlying collection*, but that is not the case for an unmodifiable collection. In the case of an unmodifiable view of a collection, any changes to the underlying collection are reflected in the unmodifiable view.

Examples of *unmodified collections* include the following:

- Unmodifiable lists created by `List.of()` and `List.copyOf()` methods (§12.2, p. 649). See also the comparison of unmodifiable lists and *list views* that are created by the `Arrays.asList()` method (§12.7, p. 660).
- Unmodifiable sets created by `Set.of()` and `Set.copyOf()` methods (p. 804).

- Unmodifiable maps created by `Map.of()`, `Map.ofEntries()`, and `Map.copyOf()` methods (p. 832).

The code below creates an *unmodifiable view* of a map whose values are mutable.

Click here to view code image

```
// Mutable courses:
StringBuilder mc1 = new StringBuilder("Java I");
StringBuilder mc2 = new StringBuilder("Java II");
StringBuilder mc3 = new StringBuilder("Java III");
StringBuilder mc4 = new StringBuilder("Java IV");

// Backing map:
Map<Integer, StringBuilder> backingMap = new HashMap<>();
backingMap.put(200, mc1); backingMap.put(300, mc2);
backingMap.put(400, mc3); backingMap.put(500, mc4);
```

Click here to view code image

```
// Unmodifiable view of a map:
Map<Integer, StringBuilder> unmodViewMap
               = Collections.unmodifiableMap(backingMap);
```

As in the case of unmodifiable collections, the code below throws an `Unsupported-OperationException` when an attempt is made to structurally change the unmodifiable view of a map.

Click here to view code image

```
 // UnsupportedOperationException at (1), (2), and (3):
unmodViewMap.put(100, new StringBuilder("Java Now"));
unmodViewMap.remove(400);
unmodViewMap.replace(200, new StringBuilder("First Java"));
```

However, changes to the backing map are reflected in the unmodifiable view:

Click here to view code image

```
backingMap.remove(200); backingMap.remove(400);
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {500=Java Complete, 300=Java II}
//  Unmodifiable view: {500=Java Complete, 300=Java II}
```

Since the values in the unmodifiable view of a map are mutable, the code below shows that changing a value of an entry in the unmodifiable view is reflected in the backing map.

Click here to view code image

```
StringBuilder mutableCourse = unmodViewMap.get(500);  // Get the course.
mutableCourse.replace(5, 8, "Complete");              // Change the course nam
```

```
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {400=Java III, 500=Java Complete, 200=Java I, 300=Java
//  Unmodifiable view: {400=Java III, 500=Java Complete, 200=Java I, 300=Java
```

Ordering Elements in Lists

In order to sort the elements of a collection by their *natural ordering* or by another *total ordering*, the elements must implement the `Comparable<E>` (§14.4, p. 761) or the `Comparator<E>` (§14.5, p. 769) interface, respectively.

The `Collections` class provides two generic static methods for sorting lists.

> Click here to view code image
>
> ```
> <E extends Comparable<? super E>> void sort(List<E> list)
> <E> void sort(List<E> list, Comparator<? super E> comparator)
> ```
>
> The first method sorts the elements in the list according to their natural ordering. The second method does the sorting according to the total ordering defined by the comparator.
>
> In addition, all elements in the list must be *mutually comparable*: The method call `e1.compareTo(e2)` (or `e1.compare(e2)` in the case of the comparator) must not throw a `ClassCastException` for any elements `e1` and `e2` in the list. In other words, it should be possible to compare any two elements in the list. Note that the second method does not require that the type parameter `E` is `Comparable<E>`.
>
> If the specified comparator is `null` then the natural ordering for the elements is used, requiring elements in this list to implement the `Comparable<E>` interface.
>
> Click here to view code image
>
> ```
> <E> Comparator<E> reverseOrder()
> <E> Comparator<E> reverseOrder(Comparator<E> comparator)
> ```
>
> The first method returns a comparator that enforces the reverse of the natural ordering. The second method reverses the total ordering defined by the comparator. Both are useful for maintaining objects in reverse-natural or reverse-total ordering in sorted collections and arrays.

The `List<E>` interface also defines an abstract method for sorting lists, with semantics equivalent to its namesake in the `Collections` class.

> Click here to view code image
>
> ```
> // Defined in the List <E> interface.
> void sort(Comparator<? super E> comparator)
> ```
>
> This list is sorted according to the total ordering defined by the specified comparator. If the specified comparator is `null` then the natural ordering for the elements is used, requiring elements in this list to implement the `Comparable<E>` interface.

This code shows how a list of strings is sorted according to different criteria. We have used the `sort()` method from the `List<E>` interface and from the `Collections` class.

Click here to view code image

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "biggest", "big", "bigger", "Bigfoot");

strList.sort(null);                                      // Natural order
strList.sort(Comparator.comparing(String::length));      // length order
strList.sort(Collections.reverseOrder());                // Reverse natural or
Collections.sort(strList, String.CASE_INSENSITIVE_ORDER);// Case insensitive o
Collections.sort(strList,                        // Reverse case insensitive o
                 Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

The output below shows the list before sorting, followed by the results from the calls to the `sort()` methods above, respectively:

Click here to view code image

```
Before sorting:                             [biggest, big, bigger, Bigfoot]
After sorting in natural order:             [Bigfoot, big, bigger, biggest]
After sorting in length order:              [big, bigger, Bigfoot, biggest]
After sorting in reverse natural order:     [biggest, bigger, big, Bigfoot]
After sorting in insensitive order:         [big, Bigfoot, bigger, biggest]
After sorting in reverse insensitive order: [biggest, bigger, Bigfoot, big]
```

It is important to note that either the element type of the list must implement the `Comparable<E>` interface or a `Comparator<E>` must be provided. The following code sorts a list of `StringBuilder` according to their natural ordering, as the class `String-Builder` implements the `Comparable<StringBuilder>` interface.

Click here to view code image

```
List<StringBuilder> sbList = new ArrayList<>();
Collections.addAll(sbList, new StringBuilder("smallest"),
                   new StringBuilder("small"), new StringBuilder("smaller"));
Collections.sort(sbList);                 // [small, smaller, smallest]
```

Below is an example of a list whose elements are not mutually comparable. Raw types are used intentionally to create such a list. Predictably, the `sort()` method throws an exception because the primitive wrapper classes do not permit interclass comparison.

Click here to view code image

```
List freakList = new ArrayList();                        // Raw types.
Collections.addAll(freakList, 23, 3.14, 10L);
freakList.sort(null);                                    // ClassCastException
```

The comparator returned by the `reverseOrder()` method can be used with *sorted* collections. The elements in the following sorted set would be maintained in descending order:

Click here to view code image

```
Set<Integer> intSet = new TreeSet<>(Collections.reverseOrder());
Collections.addAll(intSet, 9, 11, -4, 1);
System.out.println(intSet);           // [11, 9, 1, -4]
```

The following utility methods in the `Collections` class apply to *any* list, regardless of whether the elements are `Comparable<E>` or not:

> ```
> void reverse(List<?> list)
> ```
>
> Reverses the order of the elements in the list.
>
> Click here to view code image
>
> ```
> void rotate(List<?> list, int distance)
> ```
>
> Rotates the elements toward the end of the list by the specified distance. A negative value for the `distance` will rotate toward the start of the list.
>
> Click here to view code image
>
> ```
> void shuffle(List<?> list)
> void shuffle(List<?> list, Random rnd)
> ```
>
> Randomly permutes the list—that is, *shuffles* the elements.
>
> Click here to view code image
>
> ```
> void swap(List<?> list, int i, int j)
> ```
>
> Swaps the elements at indices `i` and `j`.

The effect of these utility methods can be limited to a sublist—that is, a segment of the list. The following code illustrates rotation of elements in a list. Note how the rotation in the sublist view is reflected in the original list.

Click here to view code image

```
// intList refers to the following list:                  [9, 11, -4, 1,
Collections.rotate(intList, 2);        // Two to the right.     [1, 7, 9, 11,
Collections.rotate(intList, -2);       // Two to the left.      [9, 11, -4, 1,
List intSublist = intList.subList(1,4);// Sublist:                 [11, -4, 1]
Collections.rotate(intSublist, -1);    // One to the left.         [-4, 1, 11]
                                       // intList is now:       [9, -4, 1, 11,
```

Searching in Collections

The `Collections` class provides two static methods for finding elements in *sorted* lists.

> Click here to view code image
>
> ```
> <E> int binarySearch(List<? extends Comparable<? super E>> list, E key)
> <E> int binarySearch(List<? extends E> list, E key,
>                      Comparator<? super E> cmp))
> ```
>
> These methods use binary search to find the index of the `key` element in the specified sorted list. The first method requires that the list is sorted according to natural ordering, whereas the second method requires that it is sorted according to the total ordering dictated by the comparator. The elements in the list and the key must also be *mutually comparable*.

Successful searches return the index of the key in the list. A non-negative value indicates a successful search. Unsuccessful searches return a negative value given by the formula `-(`*insertion point* + 1`)`, where *insertion point* is the index where the key would have been, had it been in the list. In the code below, the return value `-3` indicates that the key would have been at index 2, had it been in the list.

Click here to view code image

```
Collections.sort(strList);
// Sorted in natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   // Successful:
out.println(Collections.binarySearch(strList, "bigfeet"));  // Unsuccessful: -
out.println(Collections.binarySearch(strList, "bigmouth")); // Unsuccessful: -
```

Proper use of the search methods requires that the list is sorted, and the search is performed according to the same sort order. Otherwise, the search results are *unpredictable*. The example below shows the results of the search when the list `str-List` above was sorted in reverse natural ordering, but was searched assuming natural ordering. Most importantly, the return values reported for unsuccessful searches for the respective keys are incorrect in the list that was sorted in reverse natural ordering.

Click here to view code image

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   //  1
out.println(Collections.binarySearch(strList, "bigfeet"));  // -1 (INCORRECT)
out.println(Collections.binarySearch(strList, "bigmouth")); // -5 (INCORRECT)
```

Searching the list in reverse natural ordering requires that an appropriate comparator is supplied during the search (as during the sorting), resulting in correct results:

Click here to view code image

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in reverse natural order:
out.println(Collections.binarySearch(strList, "bigger",
                                     Collections.reverseOrder())); //  1
out.println(Collections.binarySearch(strList, "bigfeet",
                                     Collections.reverseOrder())); // -3
out.println(Collections.binarySearch(strList, "bigmouth",
                                     Collections.reverseOrder())); // -1
```

The following methods search for *sublists*:

Click here to view code image

```
int indexOfSubList(List<?> source, List<?> target)
int lastIndexOfSubList(List<?> source, List<?> target)
```

These two methods find the first or the last occurrence of the `target` list in the `source` list, respectively. They return the starting position of the `target` list in the `source` list. The methods are applicable to lists of *any* type.

The following methods find the maximum and minimum elements in a collection:

Click here to view code image

```
<E extends Object & Comparable<? super E>>
    E max(Collection<? extends E> c)
<E> E max(Collection<? extends E> c, Comparator<? super E> cmp)
<E extends Object & Comparable<? super E>>
    E min(Collection<? extends E> c)
<E> E min(Collection<? extends E> cl, Comparator<? super E> cmp)
```

> The one-argument methods require that the elements have a natural order-ing—that is, are `Comparable<E>`. The other methods require that the elements have a total ordering enforced by the comparator. Calling any of the methods with an empty collection as a parameter results in a `NoSuchElementException`.
>
> The time for the search is proportional to the size of the collection.
>
> These methods are analogous to the methods `first()` and `last()` in the `SortedSet<E>` class (p. 810), and the methods `firstKey()` and `lastKey()` in the `SortedMap<K, V>` class (p. 845).

Replacing Elements in Collections

Both the `List<E>` interface and the `Collections` class define a `replaceAll()` method to replace elements in a collection, but they operate differently.

> Click here to view code image
>
> ```
> // Defined in the List<E> interface.
> void replaceAll(UnaryOperator<E> operator)
> ```
>
> With this method, each element is replaced with the result of applying the specified `operator` to that element of this list. Unary operators are covered in §13.10, p. 720.
>
> Click here to view code image
>
> ```
> // Defined in the Collections class.
> <E> boolean replaceAll(List<E> list, E oldVal, E newVal) // Collections
> ```
>
> Replaces all elements equal to `oldVal` with `newVal` in the `list`; it returns `true` if the `list` was modified.

The following code snippet illustrates how the `replaceAll()` method of the `List<E>` interface can be used to apply a `UnaryOperator` to each element of the list. The lambda expression at (1a) is executed for each string in the list `strList`, replacing each element with an uppercase version of the string. Equivalent method reference is used for the same purpose at (1b).

Click here to view code image

```
// Before: [biggest, big, bigger, Bigfoot]
strList.replaceAll(str -> str.toUpperCase());      // (1a)
strList.replaceAll(String::toUpperCase);           // (1b)
```

```
// After: [BIGGEST, BIG, BIGGER, BIGFOOT]
```

In contrast, the `replaceAll()` method of the `Collections` class can be used to replace all occurrences of a specific value in the collection with a new value. In the list `palindromes` of strings, the occurrence of the string `"road"` is replaced by the string `"anna"` in the method call below.

Click here to view code image

```
// Before: [eye, level, radar, road]
Collections.replaceAll(palindromes, "road", "anna");
// After: [eye, level, radar, anna]
```

The majority of the methods found in the `Collections` class that replace the elements of a collection operate on a `List`, while one method operates on arbitrary `Collection`s. They all change the contents of the collection in some way.

> Click here to view code image
>
> ```
> <E> boolean addAll(Collection<? super E> collection, E... elements)
> ```
>
> Adds the specified `elements` to the specified `collection`. This is a convenient method for loading a collection with individually specified elements or an array. The method is annotated with `@SafeVarargs` because of the variable arity parameter. The annotation suppresses the heap pollution warning in its declaration and the unchecked generic array creation warning at the call sites.
>
> Click here to view code image
>
> ```
> <E> void copy(List<? super E> destination, List<? extends E> source)
> ```
>
> Adds the elements from the `source` list to the `destination` list. Elements copied to the `destination` list will have the same index as in the `source` list. The `destination` list cannot be shorter than the `source` list. If it is longer, the remaining elements in the `destination` list are unaffected.
>
> Click here to view code image
>
> ```
> <E> void fill(List<? super E> list, E element)
> ```
>
> Replaces all of the elements of the `list` with the specified `element`.
>
> Click here to view code image
>
> ```
> <E> List<E> nCopies(int n, E element)
> ```

> Creates an immutable list with `n` copies of the specified `element`. The *same* reference value of the specified element is saved in the list for *all* references.

The `addAll()` method is a convenient method for loading an *existing* collection with a *few* individually specified elements or an array of small size. Several examples of its usage can be found in this chapter. The array passed should be an array of objects. Note also the autoboxing of the `int` values specified at (1) and (2). The `addAll()` method does not allow primitive arrays as a variable arity argument, as attempted at (3).

Click here to view code image

```
List<Integer> intList = new ArrayList<>();            // []
Collections.addAll(intList, 9, 1, 1);                 // (1) Varargs
// After: [9, 1, 1]
Collections.addAll(intList, new Integer[] {1, 1, 9}); // (2) An array of Integ
// After: [9, 1, 1, 1, 1, 9]
Collections.addAll(intList, new int[] {1, 9, 1});     // (3) Compile-time erro
```

As we can see at (2), the collection returned by the `Collections.addAll()` method is *not* of fixed size as more elements can be added to it. This is in contrast to the list returned by the `Array.asList()` method (§12.7, p. 659). However, the `addAll()` method of the `Collection<E>` interface can be used for adding an arbitrary collection to an existing collection.

When using the `Collections.copy()` method to copy elements from the source list to the destination list, the elements are copied to the *same* positional index in the destination list as they were in the source list.

Click here to view code image

```
List<String> dest = new ArrayList<>();
Collections.addAll(dest, "one", "two", "three", "four");// [one, two, three, f
List<String> src = new ArrayList<>();
Collections.addAll(src, "I", "II", "III");              // [I, II, III]
Collections.copy(dest, src);                            // [I, II, III, four]
```

All elements of a list can be replaced with the *same* element, as shown at (1):

Click here to view code image

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "liberty", "equality", "fraternity");
// Before: [liberty, equality, fraternity]
Collections.fill(strList, "CENSORED");                  // (1)
// After: [CENSORED, CENSORED, CENSORED]
```

Earlier we saw usage of the `Collections.nCopies()` method in Example 15.10 and Example 15.11. The `for(;;)` loop below

Click here to view code image

```
for (int i = 0; i < 5; ++i)
  System.out.printf("%d %s%n", i, Collections.nCopies(i, "*"));
```

prints the following:

```
0 []
1 [*]
2 [*, *]
3 [*, *, *]
4 [*, *, *, *]
```

15.12 The `Arrays` Class

In this section we look at some selected utility methods from the `java.util.Arrays` class, in particular, for sorting, searching, and comparing arrays.

Creating *list views of arrays* using the `Arrays.asList()` method is covered in §12.7, p. 660, and can be compared with *unmodifiable lists* created using the factory methods `List.of()` and `List.copyOf()`.

Using the overloaded static method `Arrays.stream()` to create streams with arrays as the data source is covered in §16.4, p. 898.

The utility methods declared in the `Arrays` *class are all* `public` *and* `static`. *Therefore, these two modifiers will be omitted in their method header declarations in this section.*

Sorting Arrays

Sorting implies ordering the elements according to some ranking criteria, usually based on the *values* of the elements. The values of numeric data types are compared and ranked by using the relational operators that define the *numerical order* of the values. Objects are typically compared according to their *natural ordering* defined by their class. The class implements the `compareTo()` method of the `Comparable<E>` interface. The ordering defined by this method is called the *natural ordering* for the objects of the class, where `obj1.compareTo(obj2)` returns the following result:

- A *positive value* if `obj1` is *greater* than `obj2`

- The value `0` if `obj1` is *equal* to `obj2`
- A *negative value* if `obj1` is *less* than `obj2`

The wrapper classes for primitive values and the `String` class implement the `compareTo()` method (§8.3, p. 432), thereby giving their objects a natural ordering. Arrays with objects of these classes can readily be sorted as the sort algorithm can take advantage of their natural ordering.

The `Arrays` class provides enough overloaded versions of the `sort()` method to sort practically any type of array. The discussion on sorting lists (p. 858) is also applicable to sorting arrays.

> Click here to view code image
>
> ```
> void sort(type[] array)
> void sort(type[] array, int fromIndex, int toIndex)
> ```
>
> The permitted *type* for elements includes `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`.
>
> These methods sort the elements in the array according to their *natural ordering*.
>
> In the case of an array of objects being passed as an argument, the *objects* must be *mutually comparable* according to the *natural ordering* defined by the `Comparable<E>` interface; that is, it should be possible to compare any two objects in the array according to their natural ordering without throwing a `ClassCastException`.
>
> Click here to view code image
>
> ```
> <E> void sort(E[] array, Comparator<? super E> cmp)
> <E> void sort(E[] array, int fromIndex, int toIndex,
>               Comparator<? super E> cmp)
> ```
>
> These two generic methods sort the array according to the *total ordering* dictated by the comparator. In particular, the methods require that the elements are mutually comparable according to this comparator.
>
> The subarray bounds, if specified in the methods above, define a half-open interval. Only elements in this interval are then sorted.
>
> The `Arrays` class also defines analogous methods with the name `parallelSort` for sorting the elements in parallel.

The experiment with a list of strings (p. 858) is repeated in the following with an array of strings, giving identical results. An array of strings is sorted according to different criteria.

Click here to view code image

```
String[] strArray =  {"biggest", "big", "bigger", "Bigfoot"};
Arrays.sort(strArray);                                     // Natural orde
Arrays.sort(strArray, Comparator.comparing(String::length));   // Length order
Arrays.sort(strArray, Collections.reverseOrder());   // Reverse natural order
Arrays.sort(strArray, String.CASE_INSENSITIVE_ORDER);// Case insensitive order
Arrays.sort(strArray,                           // Reverse case insensitive order
            Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

The output below shows the array before sorting, followed by the results from the calls to the `Arrays.sort()` methods above, respectively:

Click here to view code image

```
Before sorting:                                  [biggest, big, bigger, Bigfoo
After sorting in natural order:                  [Bigfoot, big, bigger, bigges
After sorting in length order:                   [big, bigger, Bigfoot, bigges
After sorting in reverse natural order:          [biggest, bigger, big, Bigfoo
After sorting in case insensitive order:         [big, Bigfoot, bigger, bigges
After sorting in reverse case insensitive order: [biggest, bigger, Bigfoot, bi
```

The examples below illustrate sorting an array of primitive values (`int`) at (1), an array of type `Object` containing mutually comparable elements (`String`) at (2), and a half-open interval in reverse natural ordering at (3). A `ClassCastException` is thrown when the elements are not mutually comparable, at (4) and (5).

Click here to view code image

```
int[] intArray = {5, 3, 7, 1};                         // int
Arrays.sort(intArray);                                 // (1) Natural order: [1, 3, 5,

Object[] objArray1 = {"I", "am", "OK"};                // String
Arrays.sort(objArray1);                                // (2) Natural order: [I, OK, a

Comparable<Integer>[] comps = new Integer[] {5, 3, 7, 1}; // Integer
Arrays.sort(comps, 1, 4, Collections.reverseOrder());// (3) Reverse natural or
                                                      //       [5, 7, 3, 1]

Object[] objArray2 = {23, 3.14, "ten"};                // Not mutually comparable
//  Arrays.sort(objArray2);                            // (4) ClassCastException

Number[] numbers = {23, 3.14, 10L};                    // Not mutually comparable
//  Arrays.sort(numbers);                              // (5) ClassCastException
```

Searching in Arrays

A common operation on an array is to search the array for a given element, called the *key*. The `Arrays` class provides enough overloaded versions of the `binarySearch()` method to search in practically any type of array that is *sorted*. The discussion on searching in lists (p. 861) is also applicable to searching in arrays.

> Click here to view code image
>
> ```
> int binarySearch(type[] array, type key)
> int binarySearch(type[] array, int fromIndex, int toIndex, type key)
> ```
>
> The permitted *type* for elements includes `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`.
>
> These methods search for `key` in the `array` where the elements are sorted according to the natural ordering of the elements.
>
> In the case where an array of objects is passed as an argument, the *objects* must be sorted in natural ordering, as defined by the `Comparable<E>` interface.
>
> These methods return the index to the key in the sorted array, if the key exists. If not, a negative index is returned, corresponding to –(*insertion point* + 1), where *insertion point* is the index of the element where the key would have been found, if it had been in the array. In case there are duplicate elements equal to the key, there is no guarantee which duplicate's index will be returned. The elements and the key must be *mutually comparable*.
>
> The bounds, if specified in the methods, define a half-open interval. The search is then confined to this interval.
>
> Click here to view code image
>
> ```
> <E> int binarySearch(E[] array, E key, Comparator<? super E> c)
> <E> int binarySearch(E[] array, int fromIndex, int toIndex, E key,
>                      Comparator<? super E> c)
> ```
>
> These two generic methods require that the array is sorted according to the total ordering dictated by the comparator. In particular, its elements are mutually comparable according to this comparator. The comparator must be equivalent to the one that was used for sorting the array; otherwise, the results are unpredictable.

An appropriate `import` statement should be included in the source code to access the `java.util.Arrays` class by its simple name. The experiment from p. 861 with a list of strings is repeated here with an array of strings, giving identical results. In the code below, the return value `-3` indicates that the key would have been found at index 2 had it been in the list.

Click here to view code image

```
Arrays.sort(strArray);
// Sorted according to natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Arrays.binarySearch(strArray, "bigger"));   // Successful:    2
out.println(Arrays.binarySearch(strArray, "bigfeet"));  // Unsuccessful: -3
out.println(Arrays.binarySearch(strArray, "bigmouth")); // Unsuccessful: -5
```

Results are unpredictable if the array is not sorted, or the ordering used in the search is not the same as the sort order. Searching in the `strArray` using reverse natural ordering when the array is sorted in natural ordering gives the wrong result:

Click here to view code image

```
out.println(Arrays.binarySearch(strArray, "bigger",
                                Collections.reverseOrder()));  //  -1 (INCORRE
```

A `ClassCastException` is thrown if the key and the elements are not mutually comparable:

Click here to view code image

```
out.println(Arrays.binarySearch(strArray, 4)); // Key: 4 => ClassCastException
```

However, this incompatibility is caught at compile time in the case of arrays with primitive values:

Click here to view code image

```
// Sorted int array (natural order): [1, 3, 5, 7]
out.println(Arrays.binarySearch(intArray, 4.5));// Key: 4.5 => Compile-time er
```

The method `binarySearch()` derives its name from the divide-and-conquer algorithm that it uses to perform the search. It repeatedly divides the remaining elements to be searched into two halves and selects the half containing the key in which to continue the search, until either the key is found or there are no more elements left to search.

Comparing Arrays

The `java.util.Arrays` class provides a rich set of static methods for comparing arrays of primitive data types and objects. In this section we only consider the basic methods for array equality, array comparison, and array mismatch. We encourage the curious reader to explore the `Arrays` class API for more flexible comparison of arrays.

Array Equality

The `Arrays.equals()` static method can be used to determine if two arrays are equal.

> Click here to view code image
>
> ```
> boolean equals(type[] a, type[] b)
> ```
>
> The permitted *type* for elements includes all *primitive types* (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`) and `Object`.
>
> Two arrays are considered *equal* if they contain the same elements in the same order—that is, they have the same length and corresponding pairs of elements are equal. Two array references are also considered equal if both are `null`.
>
> Two primitive values `v1` and `v2` are considered *equal* if `new` *Wrapper-Class*`(v1).equals(new` *WrapperClass*`(v2))`, where *WrapperClass* is the wrapper class corresponding to their primitive data type.
>
> Two objects `obj1` and `obj2` are considered *equal* if `Objects.equals(obj1, obj2)`.

Given two arrays `fruitBasketA` and `fruitBasketB` of strings, shown below graphically, we can conclude that they are equal, since they have the same length and corresponding pairs of fruits are equal.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Equals: true
```

However, the arrays `fruitBasketA` and `fruitBasketC` below would not be considered equal. Although they have the same length and the same fruits, their corresponding pairs of fruits are *not* equal. The first index where this occurs is index 2.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Equals: false
```

Obviously, the two arrays `fruitBasketA` and `fruitBasketE` that have different lengths are not equal, as we can see below:

Click here to view code image

```
Index                 0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Equals: false
```

The following code demonstrates the examples provided above, where `System.out` is statically imported:

Click here to view code image

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
...
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketB)); // true
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketC)); // false
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketE)); // false
```

According to the `equals()` method, two `null` arrays are equal. However, the first statement below will not compile. The compiler issues an error, as the method call matches many overloaded methods named `equals` in the `Arrays` API. A cast is necessary to disambiguate the method call, as shown in the second statement.

Click here to view code image

```
out.println(Arrays.equals(null, null));              // Ambiguous method call.
                                                      //   Compile-time error!
out.println(Arrays.equals((String[]) null, null));   // true
```

Array Comparison

The `compare()` method of the `Comparable<E>` interface allows two objects to be compared. The comparison relationship is generalized by the `Arrays.compare()` static method to *lexicographically compare* arrays.

Click here to view code image

```
int compare(type[] a, type[] b)
<T extends Comparable<? super T>> int compare(T[] a, T[] b)
```

The permitted *type* for elements includes all *primitive types* (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`).

These methods return the value 0 if the first and second arrays

> are equal; a value less than 0 if the first array is *lexicographically less than* the second array; and a value greater than 0 if the first array is *lexicographically greater than* the second array.
>
> The second method only permits arrays whose objects implement the `compare-To()` method of the `Comparable<E>` interface.
>
> Two `null` array references are considered equal, and a `null` array reference is considered lexicographically less than a non-`null` array reference.

Consider the case where the two arrays `diceA` and `diceD` that represent the result of rolling a dice a fixed number of times. The arrays have a *prefix* (`[5, 2]`) which is common to both arrays. At index 2, `diceA[2]` (value `6`) is *greater than* `diceD[2]` (value `3`). In this case, the `compare()` method returns a *positive* value to indicate that array `diceA` is *lexicographically greater* than array `diceD`. Note that the index 2 is equal to the length of the prefix `[5, 2]` and it is a valid index in both arrays. The prefix [5, 2] is called the *common prefix* of `diceA` and `diceD`.

```
Index      0  1  2  3
diceA ==> [5, 2, 6, 3]
diceD ==> [5, 2, 3]
Common prefix: [5,2]
Compare value: 1
```

In the example below, the `compare()` method returns a *negative* value to indicate that `diceA` is *lexicographically less* than `diceE` because `diceA[1]` (value `2`) is less than `diceE[1]` (value `6`), where the common prefix `[5]` has length 1.

```
Index      0  1  2  3
diceA ==> [5, 2, 6, 3]
diceE ==> [5, 6]
Common prefix: [5]
Compare value: -1
```

If the arrays share a common prefix, as in the examples above, the lexicographical comparison is determined by the corresponding pair of values at the index given by the length of the common prefix. The length of a common prefix is always greater than or equal to 0 and less than the minimum length of the two arrays. By this definition, the length of a common prefix can never be equal to the length of the two arrays. A common prefix of length 0 means that the elements at index 0 from each array determine the comparison result.

Where the prefix is equal to one of the arrays, we need to consider the lengths of the arrays:

- If the lengths are equal, as shown in the example below, the arrays must be lexicographically equal and the `compare()` method returns the value `0`.

  ```
  Index       0   1   2   3
  diceA ==> [5, 2, 6, 3]
  diceB ==> [5, 2, 6, 3]
  Prefix: [5, 2, 6, 3]
  Compare value: 0
  ```

- If the lengths are *not* equal, as shown in the example below, lexicographical comparison is based on the lengths of the arrays. The longer array (`diceC`, length 4) is *lexicographically greater* than the shorter array (`diceD`, length 3). The `compare()` method returns a *positive* value. Note that the length of the prefix, 3 in this example, is a valid index in the longer array, `diceC`, but not in the shorter array, `diceD`.

  ```
  Index       0   1   2   3
  diceC ==> [5, 2, 3, 6]
  diceD ==> [5, 2, 3]
  Proper prefix: [5, 2, 3]
  Compare value: 1
  ```

  In the example above, the shorter array `diceD` is a prefix for the longer array `diceC`. Such a prefix is called a *proper prefix*—that is, it is equal to one of the arrays, where the lengths of the arrays are different.

The following code can be used to demonstrate lexicographic comparison of arrays, where `System.out` is statically imported:

Click here to view code image

```
int[] diceA = { 5, 2, 6, 3 };
int[] diceB = { 5, 2, 6, 3 };
int[] diceC = { 5, 2, 3, 6 };
int[] diceD = { 5, 2, 3 };
int[] diceE = { 5, 6 };
...
out.println("Compare value: " + Arrays.compare(diceA, diceD)); // 1
out.println("Compare value: " + Arrays.compare(diceA, diceE)); // -1
out.println("Compare value: " + Arrays.compare(diceA, diceB)); // 0
out.println("Compare value: " + Arrays.compare(diceC, diceD)); // 1
```

Array Mismatch

The `Arrays.mismatch()` static method returns the *first* index where a mismatch occurs between two arrays. If there is no mismatch, the arrays are equal, and the value `-1` is returned. The index returned is determined by the prefix of the arrays. The discussion on common and proper prefixes in the subsection on comparing arrays is highly relevant for determining mismatch (p. 869).

> Click here to view code image
>
> ```
> int mismatch(type[] a, type[] b)
> ```
>
> The permitted *type* for elements includes all *primitive types* (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`) and `Object`.
>
> This method returns the *index* of the *first* mismatch between two arrays, where `0 <=` *index* `<= Math.min(a.length, b.length)`. Otherwise, it returns `-1` if no mismatch is found.
>
> The method throws a `NullPointerException` if either of the arrays is `null`.

The examples below illustrate how the mismatch is determined.

The two arrays `fruitBasketA` and `fruitBasketC` have a *common prefix* `[oranges, apples]`. Mismatch therefore occurs at the index given by the length of the prefix (2). The elements `fruitBasketA[2]` and `fruitBasketC[2]` mismatch—that is, the element strings are not equal. The `mismatch()` method returns the value `2`, the same as the length of the common prefix.

Click here to view code image

```
Index                   0         1         2      3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Common prefix: [oranges, apples]
Mismatch index: 2
```

In the following example, the common prefix is empty. Its length is 0. The elements at index 0, `fruitBasketA[0]` and `fruitBasketG[0]`, mismatch. The index `0` is returned.

Click here to view code image

```
Index                   0         1         2      3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketG ==> [apples]
Common prefix: []
Mismatch index: 0
```

In the next example, the array lengths are equal and the prefix is the same as both arrays. There is no mismatch between the two arrays. The value `-1` is returned.

Click here to view code image

```
Index                 0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Prefix: [oranges, apples, plums, kiwi]
Mismatch value: -1
```

In the example below, the arrays have a *proper prefix*, `[oranges, apples]`, as the prefix is the same as one of the arrays and the arrays have different lengths. Mismatch therefore occurs at the index given by the length of the proper prefix (2) in the longer array, `fruitBasketA`. Index 2 is only valid in this array. The `mismatch()` method returns the index `2`.

Click here to view code image

```
Index                 0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Proper prefix: [oranges, apples]
Mismatch index: 2
```

The following code can be used to demonstrate the index value returned by the `mismatch()` method in the examples above, where `System.out` is statically imported:

Click here to view code image

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
String[] fruitBasketG = { "apples" };
...
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketC)); /
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketG)); /
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketB)); /
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketE)); /
```

Miscellaneous Utility Methods in the `Arrays` Class

The methods `toString()` and `fill()` (Example 15.10, Example 15.11) have previously been used in this chapter.

Click here to view code image

```
String toString(type[] array)
String deepToString(Object[] array)
```

> Return a text representation of the contents (or "deep contents") of the specified array. The first method calls the `toString()` method of the element if this element is not an array, but calls the `Object.toString()` method if the element is an array—that is, it creates a *one-dimensional* text representation of an array. The second method *goes deeper* and creates a multidimensional text representation for an arrays of arrays—that is, arrays that have arrays as elements. The *type* can be any primitive type.
>
> Click here to view code image

```
void fill(type[] array, type value)
void fill(type[] array, int fromIndex, int toIndex, type value)
```

> In these methods, the *type* can be any primitive type, or `Object`. The methods assign the specified `value` to each element of the specified array or the subarray given by the half-open interval whose index bounds are specified.

The code below illustrates how the `Arrays.fill()` method can be used. Each element in the local array `bar` is assigned the character `'*'` using the `fill()` method, and the array is printed.

Click here to view code image

```
for (int i = 0; i < 5; ++i) {
  char[] bar = new char[i];
  Arrays.fill(bar, '*');
  System.out.printf("%d %s%n", i, Arrays.toString(bar));
}
```

Output from the `for(;;)` loop:

```
0 []
1 [*]
2 [*, *]
3 [*, *, *]
4 [*, *, *, *]
```

The `Arrays.toString()` method has been used in many examples to convert the contents of an array to a text representation. The `Arrays.deepToString()` method can be used to convert the contents of a *multidimensional* array to a text representation, where each element that is an array is also converted to a text representation.

Click here to view code image

```
char[][] twoDimArr = new char[2][2];
```

```
Arrays.fill(twoDimArr[0], '*');
Arrays.fill(twoDimArr[1], '*');
System.out.println(Arrays.deepToString(twoDimArr));    // [[*, *], [*, *]]
```

In addition to initializing an array in an array initialization block, the following overloaded `setAll()` methods in the `Arrays` class can be used to initialize an existing array.

> Click here to view code image
>
> ```
> <T> void setAll(T[] array, IntFunction<? extends T> generator)
> void setAll(int[] array, IntUnaryOperator generator)
> void setAll(long[] array, IntToLongFunction generator)
> void setAll(double[] array, IntToDoubleFunction generator)
> ```
>
> Set all elements of the specified array, using the `provided` generator function to compute each element.

The code below illustrates how each element of an existing array can be initialized to the result of applying a function.

Click here to view code image

```
String[] strArr = new String[5];
Arrays.setAll(strArr, i -> "Str@" + i); // [Str@0, Str@1, Str@2, Str@3, Str@4]

int[] intArr = new int[4];
Arrays.setAll(intArr, i -> i * i);      // [0, 1, 4, 9]
```

Review Questions

15.20 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class WhatIsThis {
  public static void main(String[] args) {
    List<StringBuilder> list = new ArrayList<>();
    list.add(new StringBuilder("B"));
    list.add(new StringBuilder("A"));
    list.add(new StringBuilder("C"));
    list.sort(Collections.reverseOrder());
    System.out.println(list.subList(1,2));
  }
}
```

Select the one correct answer.

a. The program will compile. When run, it will print `[B]`.

b. The program will compile. When run, it will print `[B, A]`.

c. The program will compile. When run, it will throw an exception.

d. The program will fail to compile.

15.21 Which of the following statements are true about the following program?

Click here to view code image

```
import java.util.*;
public class InitOnly {
  public static void main(String[] args) {
    List<String> list1 = new ArrayList<>();
    Collections.addAll(list1, "Hi", "Hello");
    Collections.addAll(list1, "Howdy");
    System.out.println(list1);                              // (1)

    List<String> list2 = new ArrayList<>(list1);
    list2 = Arrays.asList("Hi", "Hello");
    list2 = Arrays.asList("Howdy");
    System.out.println(list2);                              // (2)
    List<String> list3 = new ArrayList<>();
    list3.addAll(list1);
    System.out.println(list3);                              // (3)

    List<String> list4 = new ArrayList<>(list2);
    System.out.println(list4);                              // (4)
  }
}
```

Select the four correct answers.

a. Line (1) will print `[Howdy]`.

b. Line (1) will print `[Hi, Hello, Howdy]`.

c. Line (2) will print `[Howdy]`.

d. Line (2) will print `[Hi, Hello, Howdy]`.

e. Line (3) will print `[Howdy]`.

f. Line (3) will print `[Hi, Hello, Howdy]`.

g. Line (4) will print `[Howdy]`.

h. Line (4) will print `[Hi, Hello, Howdy]`.

15.22 Which of the following statements are true about the following program?

Click here to view code image

```
import java.util.Arrays;
public class GetThatIndex {
  public static void main(String[] args) {

    if (args.length != 1) return;
    printIndex(args[0]);
  }

  public static void printIndex(String key) {
    String[] strings = {"small", "smaller", "smallest", "tiny"};
    System.out.println(Arrays.binarySearch(strings, key));
  }
}
```

Select the two correct answers.

a. The largest value ever printed by the `printIndex()` method is `3`.

b. The largest value ever printed by the `printIndex()` method is `4`.

c. The largest value ever printed by the `printIndex()` method is `5`.

d. The smallest value ever printed by the `printIndex()` method is `0`.

e. The smallest value ever printed by the `printIndex()` method is `-4`.

f. The smallest value ever printed by the `printIndex()` method is `-5`.

g. The smallest value ever printed by the `printIndex()` method is `-3`.

15.23 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q36 {
  public static void main(String[] args) {
    String s1 = "London";
    String s2 = "Bergen";
    String s3 = "Saratov";
    Deque<String> trip = new ArrayDeque<>();
    String r1 = trip.pollFirst();
    trip.offerFirst(s1);
    trip.offerFirst(s2);
    String r2 = trip.pollFirst();
    String r3 = trip.peekFirst();
    trip.offerLast(s3);
    trip.offerLast(s1);
    String r4 = trip.pollLast();
    String r5 = trip.peekLast();
    System.out.println(r1 + " " + r2 + " " + r3 + " " + r4 + " " + r5);
  }
}
```

What is the result?
Select the one correct answer.

a. An `UnsupportedOperationException` is thrown at runtime.

b. A `NullPointerException` is thrown at runtime.

c. `null Bergen London London` Saratov

d. `null Bergen Bergen London London`

e. `Bergen London London Saratov`

f. `Bergen Bergen London London`

15.24 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q37 {
  public static void main(String[] args) {
    Integer x = 1, y = 2, z = 3;
    Set<Integer> coordinates = new TreeSet<>();
    coordinates.add(x);
    coordinates.add(y);
    coordinates.add(y);
    coordinates.add(z);
    coordinates.remove(x);
    System.out.print(coordinates);
  }
}
```

What is the result?

Select the one correct answer.

a. `[1, 2, 3]`
b. `[2, 3]`
c. `[2, 2, 3]`
d. `[1, 2, 2, 3]`
e. The program will throw an exception at runtime.
f. The program will fail to compile.

15.25 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q38 {
  public static void main(String[] args) {
    List<Integer> prices = new ArrayList<>();
    prices.add(1);
    prices.add(2);
    prices.add(2, null);
    prices.add(3, 3);
    prices.add(2, 4);
    prices.set(2, 3);
    prices.remove(2);
    prices.add(2, 2);
    System.out.print(prices);
  }
}
```

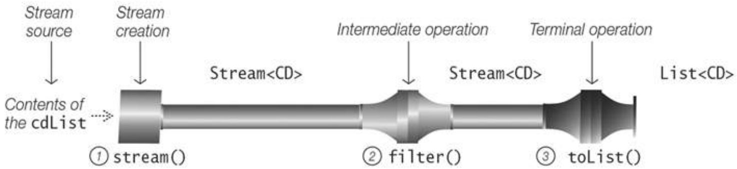
What is the result?

Select the one correct answer.

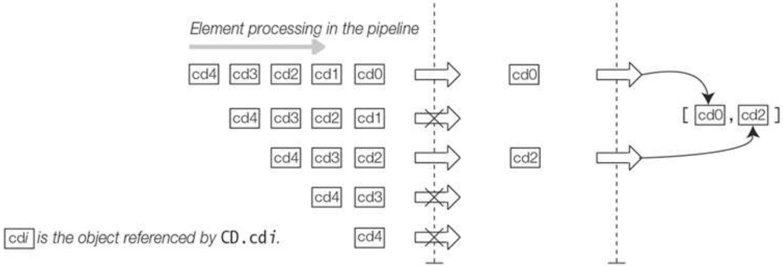
a. `[1, 2, null, 3]`
b. `[1, 2, 3]`
c. `[1, 2, 2, 3]`
d. `[1, 2, 2, null, 3]`
e. `[1, 2, null, null]`
f. `[1, null, 2, 2, 3]`
g. The program will throw an exception at runtime.
h. The program will fail to compile.

```
// Query to create a list of all CDs with pop music.
List<CD> cdList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd3, CD.cd4);
List<CD> popCDs = cdList
①  .stream()                                    // Stream creation.
②  .filter(CD::isPop)                           // Intermediate operation.
③  .toList();                                   // Terminal operation.
```

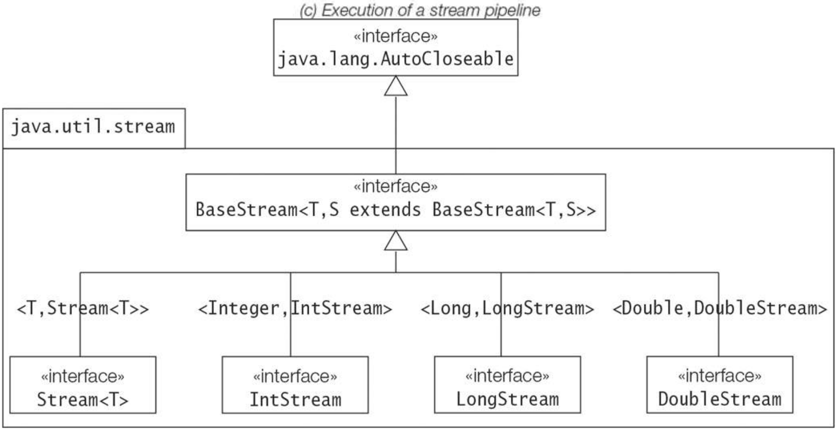
(a) A query for data processing with streams


Stream source
Stream creation
Intermediate operation
Terminal operation
Stream<CD>
Stream<CD>
List<CD>
Contents of the cdList
① stream()
② filter()
③ toList()

(b) A stream pipeline

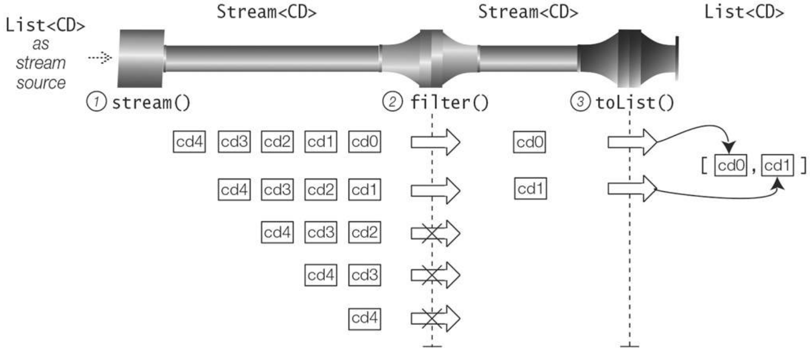

Element processing in the pipeline
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd4 cd3
cd4
cd0
cd2
[cd0 , cd2]
cdi is the object referenced by CD.cdi.

(c) Execution of a stream pipeline


«interface»
java.lang.AutoCloseable
java.util.stream
«interface»
BaseStream<T,S extends BaseStream<T,S>>
<T,Stream<T>>
<Integer,IntStream>
<Long,LongStream>
<Double,DoubleStream>
«interface»
Stream<T>
«interface»
IntStream
«interface»
LongStream
«interface»
DoubleStream

```
  // Query 1: Find CDs whose titles are in the set of popular CD titles.
  Set<String> popularTitles = Set.of("Java Jive", "Java Jazz", "Java Jam");
  List<CD> popularCDs1 = CD.cdList
①     .stream()
②     .filter(cd -> popularTitles.contains(cd.title()))
③     .toList();
```

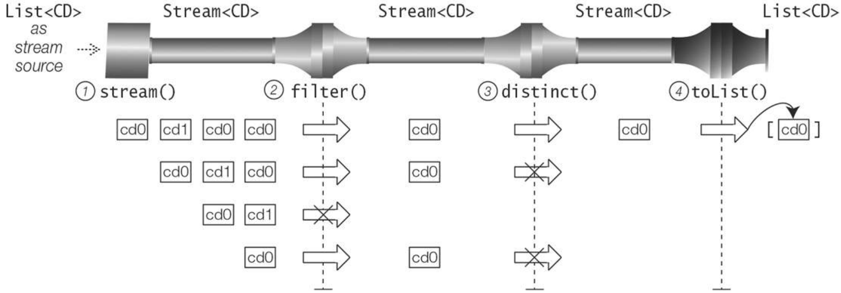
(a) Query to filter stream elements


List<CD>
as
stream
source
Stream<CD>
Stream<CD>
List<CD>
① stream()
② filter()
③ toList()
cd4 cd3 cd2 cd1 cd0
cd0
[cd0 , cd1]
cd4 cd3 cd2 cd1
cd1
cd4 cd3 cd2
cd4 cd3
cd4

(b) Execution of stream pipeline

```
  // Query 2: Create a list of unique CDs with pop music.
  List<CD> miscCDList = List.of(CD.cd0, CD.cd0, CD.cd1, CD.cd0);
  List<CD> uniquePopCDs1 = miscCDList
①     .stream()
②     .filter(CD::isPop)
③     .distinct()
④     .toList();
```

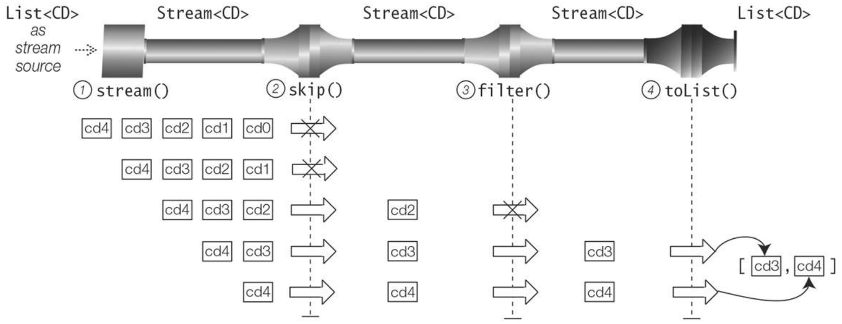
(a) Selecting distinct elements in a stream


List<CD>
as
stream
source
Stream<CD>
Stream<CD>
Stream<CD>
List<CD>
① stream()
② filter()
③ distinct()
④ toList()
cd0 cd1 cd0 cd0
cd0
cd0
[cd0]
cd0 cd1 cd0
cd0
cd0 cd1
cd0
cd0

(b) Execution of stream pipeline

```
   // Query 3a: Create a list of jazz CDs, after skipping the first two CDs.
   List<CD> jazzCDs1 = CD.cdList
①      .stream()
②      .skip(2)
③      .filter(CD::isJazz)
④      .toList();
```

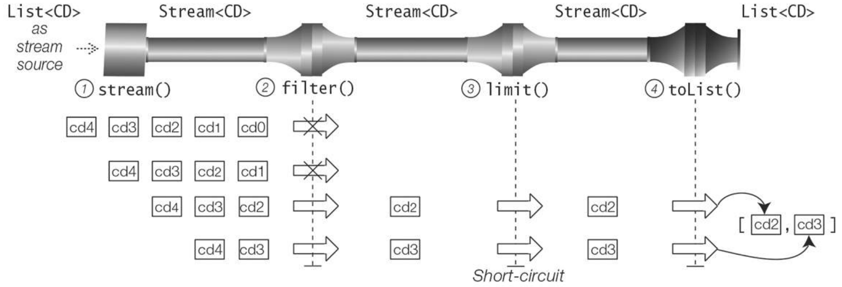
(a) *Skipping elements in a stream*


List<CD>
as
stream
source
Stream<CD>
Stream<CD>
Stream<CD>
List<CD>
① stream()
② skip()
③ filter()
④ toList()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd2
cd4 cd3
cd3
cd3
cd4
cd4
cd4
[cd3 , cd4]

(b) *Execution of stream pipeline*

```
  // Query 4: Create a list with the first 2 CDs that were released in 2018.
  List<CD> twoFirstCDs2018 = CD.cdList
①     .stream()
②     .filter(cd -> cd.year().equals(Year.of(2018)))
③     .limit(2)
④     .toList();
```

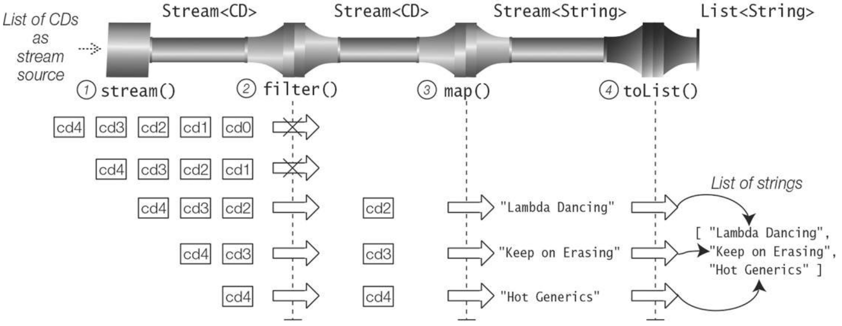
(a) *Truncating a stream*


List<CD>
as
stream
source
Stream<CD>
Stream<CD>
Stream<CD>
List<CD>
① stream()
② filter()
③ limit()
④ toList()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd2
cd2
cd4 cd3
cd3
cd3
[cd2 , cd3]
Short-circuit

(b) *Execution of stream pipeline*

```
   // Query: Create a list of CD titles released in 2018.
   List<String> listOfCDNames = CD.cdList
①      .stream()                                             // Stream<CD>
②      .filter(cd -> cd.year().equals(Year.of(2018)))       // Stream<CD>
③      .map(CD::title)                                       // Stream<String>
④      .toList();                                            // List<String>
```

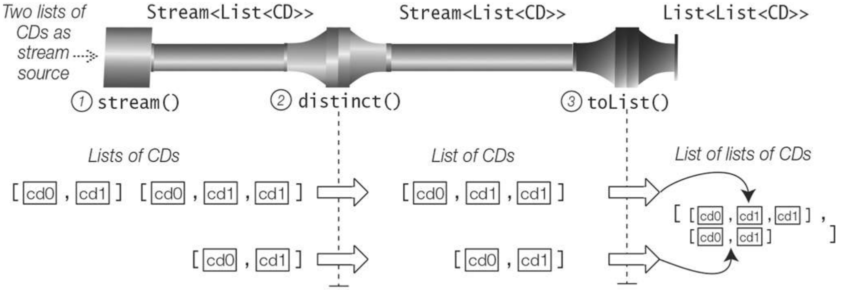
(a) *Query using the* `Stream.map()` *method*


List of CDs
as
stream
source
Stream<CD>
Stream<CD>
Stream<String>
List<String>
① stream()
② filter()
③ map()
④ toList()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd2
"Lambda Dancing"
cd4 cd3
cd3
"Keep on Erasing"
cd4
cd4
"Hot Generics"
List of strings
["Lambda Dancing",
"Keep on Erasing",
"Hot Generics"]

(b) *Execution of stream pipeline*

```
// Query: Create a list of unique CDs from two given lists of CDs.
List<List<CD>> listOfListOfCDs =
①  Stream.of(cdList1, cdList2)                          // Stream<List<CD>>
②         .distinct()                                   // Stream<List<CD>>
③         .toList());                                   // List<List<CD>>
```

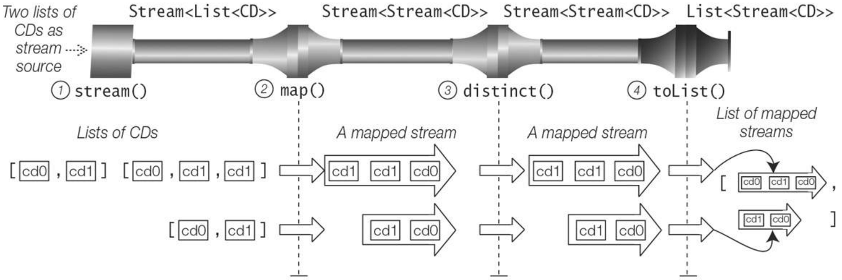
(a) Incorrect solution using the `Stream.distinct()` method


Two lists of CDs as stream source
Stream<List<CD>>
Stream<List<CD>>
List<List<CD>>
① stream()
② distinct()
③ toList()
Lists of CDs
List of CDs
List of lists of CDs
[cd0 , cd1] [cd0 , cd1 , cd1]
[cd0 , cd1 , cd1]
[[cd0 , cd1 , cd1] ,
[cd0 , cd1]]
[cd0 , cd1]
[cd0 , cd1]

(b) Stream pipeline

```
// Query: Create a list of unique CDs from two given lists of CDs.
List<Stream<CD>> listOfStreamOfCD =
①    Stream.of(cdList1, cdList2)                      // Stream<List<CD>>
②          .map(List::stream)                          // Stream<Stream<CD>>
③          .distinct()                                 // Stream<Stream<CD>>
④          .toList();                                  // List<Stream<CD>>
```

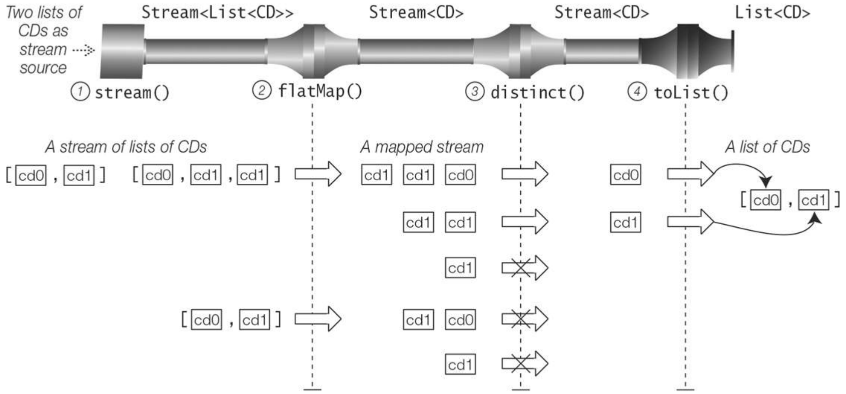
(a) Incorrect solution using the `Stream.map()` method


Two lists of CDs as stream source
Stream<List<CD>>
Stream<Stream<CD>>
Stream<Stream<CD>>
List<Stream<CD>>
① stream()
② map()
③ distinct()
④ toList()
Lists of CDs
A mapped stream
A mapped stream
List of mapped streams
[cd0 , cd1] [cd0 , cd1 , cd1]
cd1 cd1 cd0
cd1 cd1 cd0
[cd0 cd1 cd0 ,
cd1 cd0]
[cd0 , cd1]
cd1 cd0
cd1 cd0

(b) Stream pipeline

```
    //Query: Create a list of unique CDs from two given lists of CDs.
    List<CD> listOfCD =
①       Stream.of(cdList1, cdList2)                           // Stream<List<CD>>
②             .flatMap(List::stream)                          // Stream<CD>
③             .distinct()                                     // Stream<CD>
④             .toList());                                     // List<CD>
```

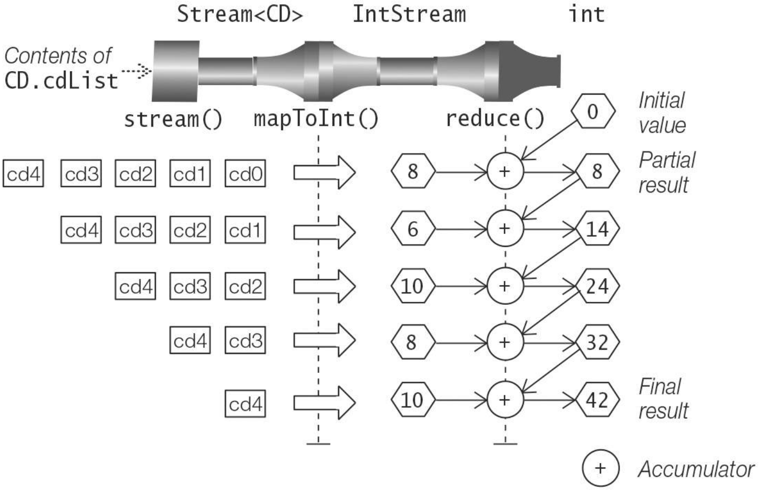
(a) Solution using the `Stream.flatMap()` method


Two lists of CDs as stream source
Stream<List<CD>>
Stream<CD>
Stream<CD>
List<CD>
① stream()
② flatMap()
③ distinct()
④ toList()
A stream of lists of CDs
A mapped stream
A list of CDs
[cd0, cd1] [cd0, cd1, cd1]
cd1 cd1 cd0
cd0
[cd0, cd1]
cd1 cd1
cd1
cd1
[cd0, cd1]
cd1 cd0
cd1

(b) Stream pipeline

```
// Query: Find the total number of CD tracks.
int totNumOfTracks = CD.cdList                               // (5)
    .stream()                                                // (6)
    .mapToInt(CD::noOfTracks)                                // (7)
    .reduce(0,                                               // (8)
            (sum, numOfTracks) -> sum + numOfTracks);        // (9)
```

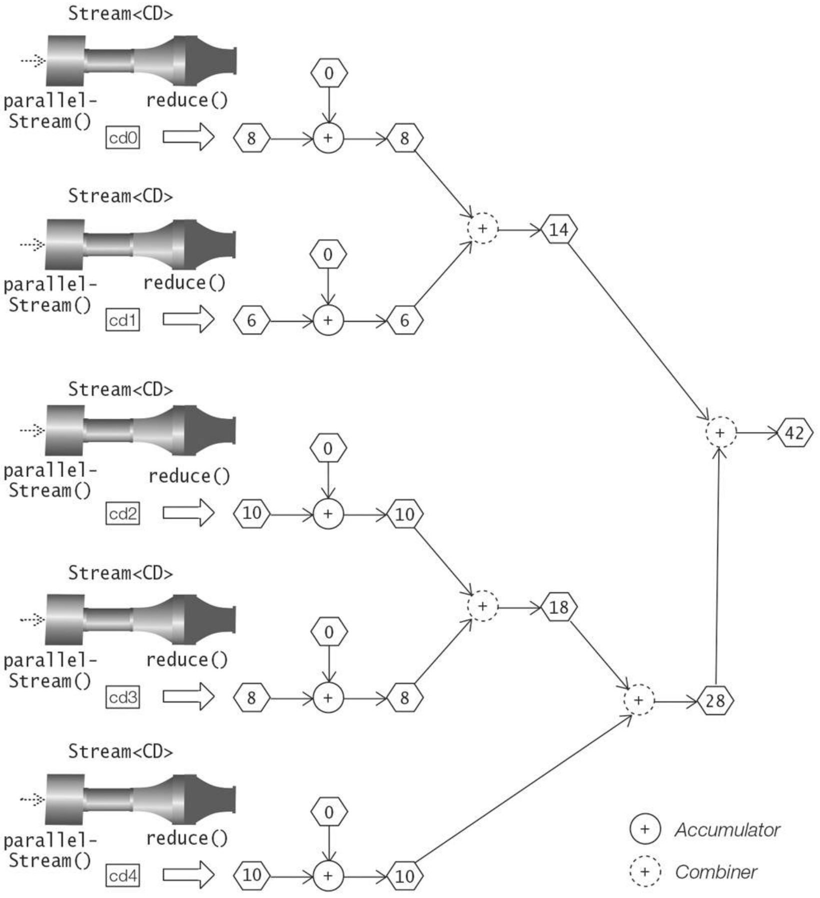
(a) Using the `reduce()` method with an initial value


Stream<CD>
IntStream
int
Contents of CD.cdList
stream()
mapToInt()
reduce()
0
Initial value
cd4 cd3 cd2 cd1 cd0
8
8
Partial result
cd4 cd3 cd2 cd1
6
14
cd4 cd3 cd2
10
24
cd4 cd3
8
32
cd4
10
42
Final result
+ Accumulator

(b) Stream pipeline

```
// Query: Find total number of CD tracks.
Integer sumTracks5 = CD.cdList                          // (17)
    .parallelStream()                                   // (18b)
    .reduce(Integer.valueOf(0),                         // (19) Initial value
            (sum, cd) -> sum + cd.noOfTracks(),         // (20) Accumulator
            (sum1, sum2) -> sum1 + sum2);               // (21) Combiner
```

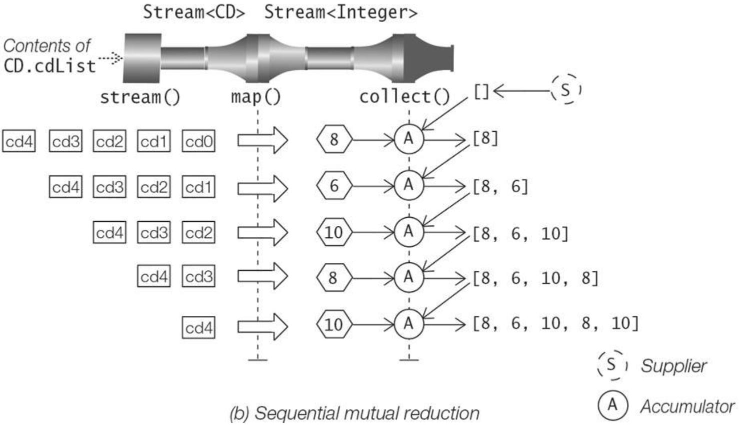
(a) Using the `Stream.reduce()` *method on a parallel stream*


Stream<CD>
parallel-Stream()
reduce()
cd0
cd1
cd2
cd3
cd4
0
8
6
10
8
10
14
18
28
42
+
Accumulator
Combiner

(b) Parallel functional reduction

```
// Query: Create a list with the number of tracks on each CD.
List<Integer> tracks = CD.cdList                              // (1)
    .stream()                                                 // (2a)
    .map(CD::noOfTracks)                                      // (3)
    .collect(() -> new ArrayList<>(),                         // (4) Supplier
             (cont, noOfTracks) -> cont.add(noOfTracks),// (5) Accumulator
             (cont1, cont2) -> cont1.addAll(cont2));    // (6) Combiner
```

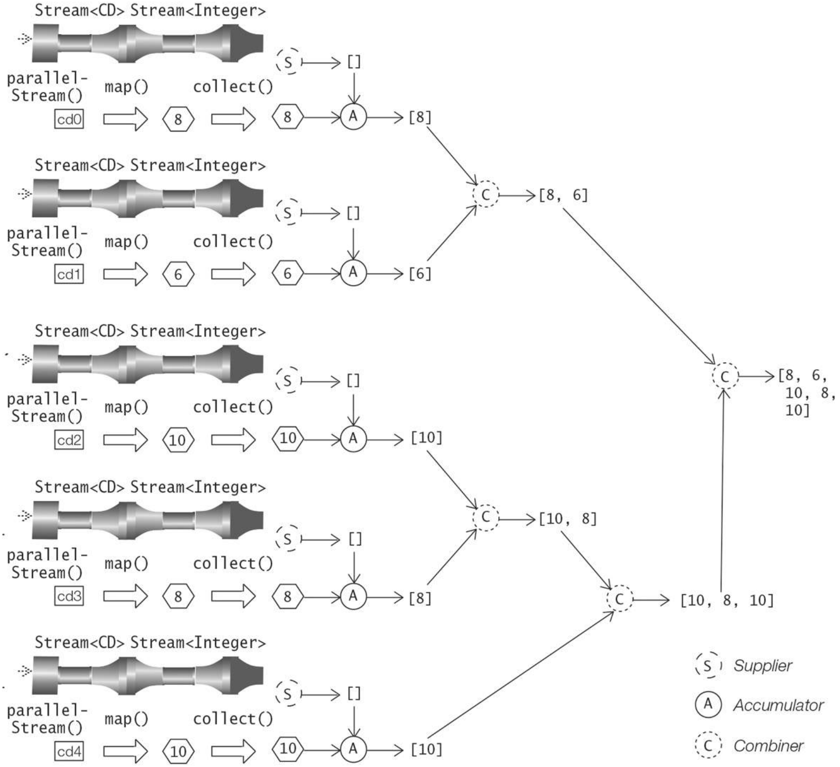
(a) Using the `Stream.collect()` *method on a sequential stream*


Stream<CD>
Stream<Integer>
Contents of
CD.cdList
stream()
map()
collect()
[]
S
cd4 cd3 cd2 cd1 cd0
8
A
[8]
cd4 cd3 cd2 cd1
6
[8, 6]
cd4 cd3 cd2
10
[8, 6, 10]
cd4 cd3
8
[8, 6, 10, 8]
cd4
10
[8, 6, 10, 8, 10]
Supplier
Accumulator

(b) Sequential mutual reduction

```
// Query: Create a list with the number of tracks on each CD.
List<Integer> tracks = CD.cdList                          // (1)
    .parallelStream()                                     // (2b)
    .map(CD::noOfTracks)                                  // (3)
    .collect(() -> new ArrayList<>(),                     // (4) Supplier
             (cont, noOfTracks) -> cont.add(noOfTracks),  // (5) Accumulator
             (cont1, cont2) -> cont1.addAll(cont2));      // (6) Combiner
```

(a) Using the `Stream.collect()` *method on a parallel stream*

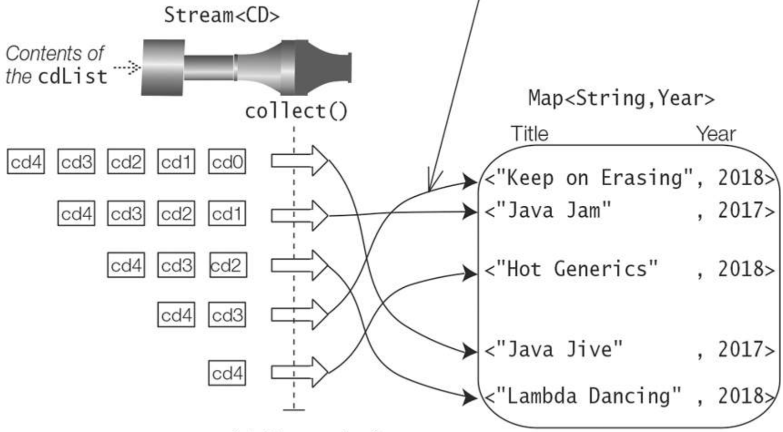


Stream<CD> Stream<Integer>
parallel-Stream()
map()
collect()
cd0
8
8
S
[]
A
[8]
cd1
6
6
[6]
C
[8, 6]
cd2
10
10
[10]
cd3
8
8
[8]
[10, 8]
[10, 8, 10]
cd4
10
10
[10]
[8, 6, 10, 8, 10]
S Supplier
A Accumulator
C Combiner

(b) Parallel mutable reduction

```
//Query: Create a map of CD titles and their release year.
Map<String, Year> mapTitleToYear = CD.cdList.stream()
          .collect(Collectors.toMap(CD::title, CD::year));
```

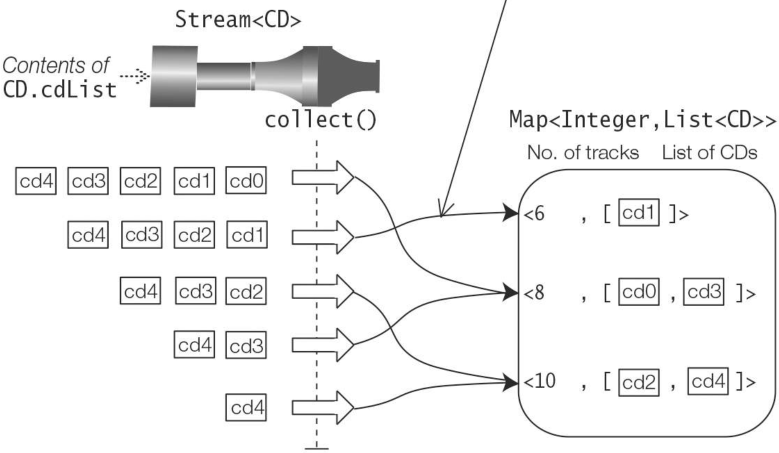
(a) Using the `Collectors.toMap()` *method*


Stream<CD>
Contents of the cdList
collect()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd4 cd3
cd4
Map<String,Year>
Title
Year
<"Keep on Erasing", 2018>
<"Java Jam" , 2017>
<"Hot Generics" , 2018>
<"Java Jive" , 2017>
<"Lambda Dancing" , 2018>

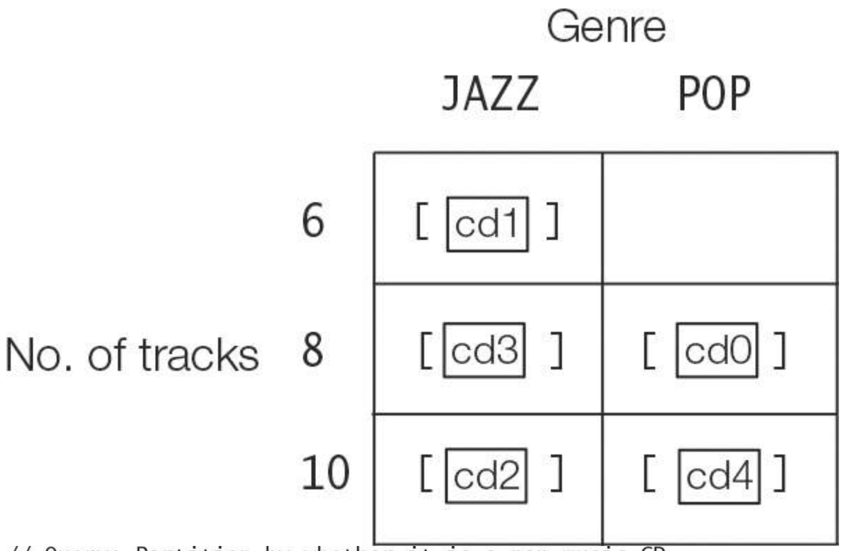
(b) Stream pipeline

```
// Query: Group by number of tracks.
Map<Integer, List<CD>> map11 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks));  // (1)
```

(a) Using the `Collectors.groupBy()` *method*

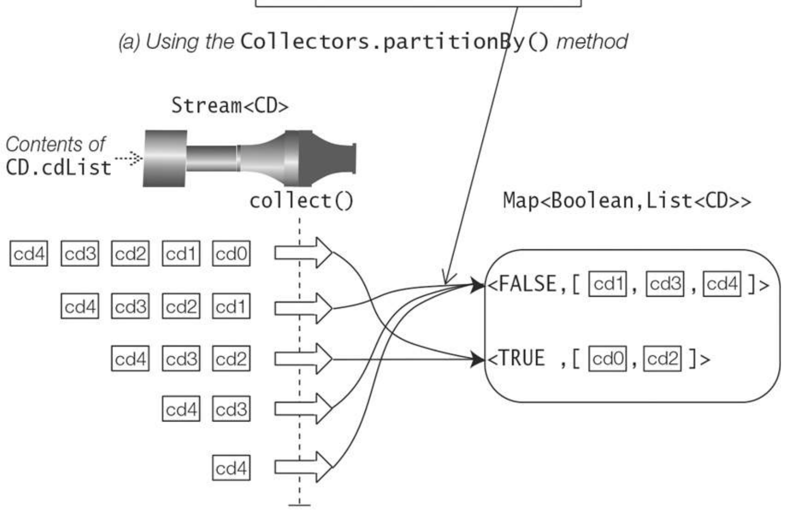

Stream<CD>
Contents of CD.cdList
collect()
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd4 cd3
cd4
Map<Integer,List<CD>>
No. of tracks
List of CDs
<6 , [cd1]>
<8 , [cd0 , cd3]>
<10 , [cd2 , cd4]>

(b) Stream pipeline


Genre
JAZZ
POP
No. of tracks
6
8
10
[cd1]
[cd3]
[cd0]
[cd2]
[cd4]

```
// Query: Partition by whether it is a pop music CD.
Map<Boolean, List<CD>> map1 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop));       // (1)
```

(a) Using the `Collectors.partitionBy()` *method*


Stream<CD>
Contents of CD.cdList
collect()
Map<Boolean,List<CD>>
cd4 cd3 cd2 cd1 cd0
cd4 cd3 cd2 cd1
cd4 cd3 cd2
cd4 cd3
cd4
<FALSE,[cd1 , cd3 , cd4]>
<TRUE ,[cd0 , cd2]>

(b) Stream pipeline

Streams 16

Chapter Topics

- Understanding the construction of a stream pipeline
- Understanding various aspects of streams: sequential or parallel, ordered or unordered, finite or infinite, and object or numeric
- Creating object streams from various sources; for example, assorted collections, arrays, strings, and I/O classes
- Creating infinite numeric streams using generator functions
- Understanding the various aspects of intermediate stream operations: stream mapping, lazy execution, short-circuit evaluation, and stateless or stateful operations
- Understanding the implications of operation order, and non-interfering and stateless behavioral parameters of intermediate stream operations
- Filtering, skipping, and examining stream elements
- Selecting distinct elements and truncating a stream
- Understanding mapping and flattening a stream
- Sorting stream elements
- Changing the execution mode of a stream and marking a stream as unordered
- Understanding interoperability between stream types
- Understanding the role of the `Optional` class
- How to create, query, filter, map, and flatten optionals
- Using numeric optionals
- Understanding the implication of invoking a terminal operation on a stream
- Applying consumer actions to elicit side effects in a stream
- Using terminal operations to match, find, and count stream elements
- Understanding functional and mutable reduction, both sequential and parallel

- Collecting stream results in lists, sets, and arrays
- Using functional reduction on numeric streams, including statistical operations
- Understanding the role of a collector in stream execution
- Collecting to a collection, list, set, map, and concurrent map
- Using a collector to join strings
- Using collectors that group and partition stream elements
- Using downstream collectors for functional reduction: counting, finding min/max, summing, averaging, and summarizing
- How to implement collectors for customized reduction
- How to use map-reduce, filtering, flat mapping, and finishing adapters for downstream collectors
- Understanding how to build and execute a parallel stream
- Understanding factors that can affect parallel stream execution
- Understanding the importance of benchmarking parallel stream execution

Java SE 17 Developer Exam O	
[6.1] Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to supply, filter, map, consume, and sort data ○ *Streams are covered in this chapter.* ○ *For lambda expressions implementing functional interfaces, see Chapter 13, p. 673.*	*§16.3, p. 884* *§16.4, p. 890* *§16.5, p. 905* *§16.7, p. 946*
[6.2] Perform decomposition, concatenation and reduction, and grouping and partitioning on sequential and parallel streams	*§16.7, p. 946* *§16.8, p. 978* *§16.9, p. 1009*
Java SE 11 Developer Exam O	
[6.2] Use Java Streams to filter, transform and process data	*§16.3, p. 884* *§16.4, p. 890* *§16.5, p. 905* *§16.7, p. 946*
[6.3] Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams	*§16.7, p. 946* *§16.8, p. 978* *§16.9, p. 1009*

The Stream API brings a new programming paradigm to Java: a *declarative* way of processing data using *streams*—expressing *what* should be done to the values and not *how* it should be done. More importantly, the API allows programmers to harness the power of multicore architectures for *parallel* processing of data.

We strongly suggest reviewing the following topics which we consider essential prerequisites for learning about streams:

- Functional-style programming (Chapter 13, p. 673); specially, functional interfaces, lambda expressions, method references, and built-in functional interfaces
- Comparing objects (Chapter 14, p. 741); in particular, the `Comparator<E>` functional interface (§14.4, p. 761)

16.1 Introduction to Streams

A *stream* allows aggregate operations to be performed on a sequence of elements. An *aggregate operation* performs a task on the stream as a whole rather than on an individual element of the stream. In the context of streams, these aggregation operations are called *stream operations*. Such operations utilize behavior parameterization implemented by functional interfaces for actions performed on the stream elements.

Examples of stream operations accepting implementation of functional interfaces include:

- Generating elements of the stream using a `Supplier`
- Converting the elements in the stream according to a mapping defined by a `Function`
- Filtering the elements in the stream according to some criteria defined by a `Predicate`
- Sorting the elements in the stream using a `Comparator`
- Performing actions for each of the elements in the stream with the help of a `Consumer`

Streams can be produced from a variety of sources. Collections and arrays are typical examples of sources for streams. The `Collection<E>` interface and the `Arrays` utility class both provide a `stream()` method that builds a stream from the elements of a collection or an array.

In the loop-based solution below, elements from the `values` list are processed using a `for(:)` loop to test whether a year is after the year 2000. The strings in the list are parsed to a `Year` object before being tested in an `if` statement.

Click here to view code image

```
// Loop-based solution:
List<String> values = List.of("2001", "1999", "2021");
for (String s : values) {
  Year y = Year.parse(s);
  if (y.isAfter(Year.of(2000))) {
    System.out.print(s + " ");                      // 2001 2021
  }
}

// Stream-based solution:
List<String> values2 = List.of("2001", "1999", "2021");
values2.stream()                                    // (1)
       .map(s -> Year.parse(s))                     // (2)
       .filter(y -> y.isAfter(Year.of(2000)))       // (3)
       .forEach(y -> System.out.print(y + " "));    // (4) 2001 2021
```

A stream-based solution for the same problem is also presented above. The `stream()` operation at (1) generates a stream based on the elements from the collection. The `map()` operation at (2) parses the string elements to a `Year` object, as defined by the lambda expression that implements the `Function` interface. The `filter()` operation at (3) performs a filtering of the elements in the stream that are after the year 2000, as defined by a lambda expression that implements the `Predicate` interface. The `forEach()` operation at (4) performs an action on each stream element, as defined by a lambda expression that implements the `Consumer` interface.

The loop-based solution specifies how the operations should be performed. The stream-based solution states what operations should be performed, qualified by the implementation of an appropriate functional interface. Stream-based solutions to many problems can be elegant and concise compared to their iteration-based counterparts.

In this chapter we will cover many stream operations in detail, as well as discover other use cases and benefits of using streams.

16.2 Running Example: The `CD` Record Class

We will use the `CD` record class and the `Genre` enum type from Example 16.1 in many of the examples throughout this chapter. The classes are intentionally kept simple for the purposes of exposition, but they will suffice in illustrating the vast number of topics covered in this chapter.

The `CD` record class declares fields for the following information about a CD: an artist name, a title, a fixed number of tracks, the year the CD was released, and a musical genre. The compiler provides the relevant get methods to access the fields in a `CD` record, but `boolean` methods are explicitly defined to determine its musical genre. The compiler also provides implementations to override the `toString()`, `hashCode()`, and `equals()` methods from the `Object` class. However, the record class provides its own implementation of the `toString()` method. The record class implements the `Comparable<CD>` interface with the following comparison order for the fields: artist name, title, number of tracks, year released, and musical genre.

It is important to note the `static` field declarations in the `CD` record class, as they will be used in subsequent examples. These include the `static` references `cd0`, `cd1`, `cd2`, `cd3`, and `cd4` that refer to five different instances of the `CD` record class. In addition, the fixed-size list `cdList` contains these five ready-made CDs, as does the array `cdArray`. The output from Example 16.1 shows the state of the CDs in the `cdList`. We recommend consulting this information in order to verify the results from examples presented in this chapter.

The simple enum type `Genre` is used to indicate the style of music on a `CD`.

Example 16.1 *The* `CD` *Example Classes*

Click here to view code image

```
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
```

Click here to view code image

```
import java.time.Year;
import java.util.Comparator;
import java.util.List;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Comparable<CD> {

  // Additional get methods:
  public boolean isPop()   { return this.genre == Genre.POP; }
  public boolean isJazz()  { return this.genre == Genre.JAZZ; }
  public boolean isOther() { return this.genre == Genre.OTHER; }
  // Provide own implementation of the toString() method.
  @Override public String toString() {
    return String.format("<%s, \"%s\", %d, %s, %s>",
        this.artist, this.title, this.noOfTracks, this.year, this.genre);
  }

  /** Compare by artist, by title, by number of tracks, by year, and by genre.
  @Override public int compareTo(CD other) {
    return Comparator.comparing(CD::artist)
                     .thenComparing(CD::title)
                     .thenComparing(CD::noOfTracks)
                     .thenComparing(CD::year)
                     .thenComparing(CD::genre)
                     .compare(this, other);
  }

  // Some ready-made CDs.
  public static final CD cd0
      = new CD("Jaav",      "Java Jive",       8,  Year.of(2017), Genre.POP);
  public static final CD cd1
      = new CD("Jaav",      "Java Jam",        6,  Year.of(2017), Genre.JAZZ);
  public static final CD cd2
      = new CD("Funkies",   "Lambda Dancing",  10, Year.of(2018), Genre.POP);
  public static final CD cd3
      = new CD("Genericos", "Keep on Erasing", 8,  Year.of(2018), Genre.JAZZ);
  public static final CD cd4
      = new CD("Genericos", "Hot Generics",    10, Year.of(2018), Genre.JAZZ);

  // A fixed-size list of CDs.
  public static final List<CD> cdList = List.of(cd0, cd1, cd2, cd3, cd4);

  // An array of CDs.
  public static final CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};
}
```

Click here to view code image

```
import java.util.List;

public final class CDAdmin {
  public static void main(String[] args) {
    List<CD> cdList = CD.cdList;
    System.out.println("     Artist    Title           No. Year Genre");
    for(int i = 0; i < cdList.size(); ++i) {
      CD cd = cdList.get(i);
      String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
          cd.artist(), cd.title(), cd.noOfTracks(),
          cd.year(), cd.genre());
      System.out.printf("cd%d: %s%n", i, cdToString);
    }
  }
}
```

Output from the program:

Click here to view code image

```
     Artist    Title           No. Year Genre
cd0: Jaav      Java Jive       8   2017 POP
cd1: Jaav      Java Jam        6   2017 JAZZ
cd2: Funkies   Lambda Dancing  10  2018 POP
cd3: Genericos Keep on Erasing 8   2018 JAZZ
cd4: Genericos Hot Generics    10  2018 JAZZ
```

16.3 Stream Basics

In this section we introduce the terminology and the basic concepts required to work with streams, and provide an overview of the Stream API.

An example of an aggregate operation is filtering the elements in the stream according to some criteria, where all the elements of the stream must be examined in order to determine the result. We saw a simple example of filtering a list in the previous section. Figure 16.1a shows an example of a *stream-based* solution for filtering a list of CDs to find all pop music CDs. A stream of CDs is created at (1). The stream is filtered at (2) to find pop music CDs. Each pop music CD that is found is accumulated into a result list at (3). Note how the method calls are chained, which is typical of processing elements in a stream. We will fill in the details as we use Figure 16.1 to introduce the basics of data processing using streams.

Figure 16.1 *Data Processing with Streams*

A stream must be built from a *data source* before operations can be performed on its elements. Streams come in two flavors: those that process *object references*, called *object streams*; and those that process numeric values, called *numeric streams*. In Figure 16.1a, the call to the `stream()` method on the list `cdList` creates an object stream of CD references with `cdList` as the data source. Building streams from data sources is explored in §16.4, p. 890.

Stream operations are characterized either as *intermediate operations* (§16.5, p. 905) or *terminal operations* (§16.7, p. 946). In Figure 16.1a, there are two stream operations. The methods `filter()` and `collect()` implement an intermediate operation and a terminal operation, respectively.

An intermediate operation always returns a new stream—that is, it transforms its *input stream* to an *output stream*. In Figure 16.1a, the `filter()` method is called at (2) on its input stream, which is the initial stream of CDs returned by the `stream()` method at (1). The method reference `CD::isPop` passed as an argument to the `filter()` method implements the `Predicate<CD>` functional interface to filter the CDs in the stream. The `filter()` method returns an output stream, which is a stream of pop music CDs.

A terminal operation either causes a side effect or computes a result. The method `collect()` at (3) implements a terminal operation that computes a result (§16.7, p. 964). This method is called on the output stream of the `filter()` operation, which is now the input stream of the terminal operation. Each CD that is determined to be a pop music CD by the `filter()` operation at (2) is accumulated into a result list by the `toList()` method at (3). The `toList()` method (p. 972) creates an empty list and accumulates elements from the input stream—in this case, CDs with pop music.

The code below splits the chain of method calls in Figure 16.1a, but produces the same result. The code explicitly shows the streams that are created and how an operation is invoked on the stream returned by the preceding operation. However, this code is a lot more verbose and not as easy to read as the code in Figure 16.1a—so much so that it is frowned upon by stream aficionados.

Click here to view code image

```
Stream<CD> stream1 = cdList.stream();          // (1a) Stream creation.
Stream<CD> stream2 = stream1.filter(CD::isPop); // (2a) Intermediate operatio
List<CD> popCDs2 = stream2.toList();           // (3a) Terminal operation.
```

Composing Stream Pipelines

Stream operations can be chained together to compose a *stream pipeline* in which stream components are specified in the following order:

- An operation on a data source for building the initial stream
- Zero or more intermediate operations to transform one stream into another
- A single mandatory terminal operation in order to execute the pipeline and produce some result or side effect

Composition into a pipeline is possible because stream creation and intermediate operations return a stream, allowing method calls to be chained as we have seen in Figure 16.1a.

The chain of method calls in Figure 16.1a forms the stream pipeline in Figure 16.1b, showing the components of the pipeline. A stream pipeline is analogous to an assembly line, where each operation depends on the result of the previous operation as parts are assembled. In a pipeline, an intermediate operation consumes elements made available by its input stream to produce elements that form its output stream. The terminal operation produces the final result from its input stream. Creating a stream pipeline can be regarded as a fusion of stream operations, where only a single pass of the elements is necessary to process the stream.

Stream operations are typically customized by behavior parameterization that is specified by functional interfaces and implemented by lambda expressions. That is why understanding built-in functional interfaces and writing method references (or their equivalent lambda expressions) is essential. In Figure 16.1a, the `Predicate` argument of the `filter()` operation implements the behavior of the `filter()` operation.

A stream pipeline formulates a query on the elements of a stream created from a data source. It expresses *what* should be done to the stream elements, and not *how* it should be done—analogous to a database query. One important advantage of composing stream pipelines is that the compiler can freely optimize the operations—for example, for parallel execution—as long as the same result is guaranteed.

Executing Stream Pipelines

Apart from the fact that an intermediate operation always returns a new stream and a terminal operation never does, another crucial difference between these two kinds of operations is the way in which they are executed. Intermediate operations use *lazy execution*; that is, they are executed on demand. Terminal operations use *eager execution*; that is, they are executed immediately when the terminal operation is invoked on the stream. This means the intermediate operation will never be executed unless a terminal operation is invoked on the output stream of the intermediate operation, whereupon the intermediate operations will start to execute and pull elements from the stream created on the data source.

The execution of a stream pipeline is illustrated in Figure 16.1c. The `stream()` method just returns the initial stream whose data source is `cdList`. The `CD` objects in `cdList` are processed in the stream pipeline in the same order as in the list, designated as `cd0`, `cd1`, `cd2`, `cd3`, and `cd4`. The way in which elements are successively processed by the operations in the pipeline is shown *horizontally* for each element. The execution of the pipeline only starts when the terminal operation `toList()` is invoked, and proceeds as follows:

1. `cd0` is selected by the `filter()` operation as it is a pop music CD, and the `collect()` operation places it in the list created to accumulate the results.
2. `cd1` is discarded by the `filter()` operation as it is not a pop music CD.
3. `cd2` is selected by the `filter()` operation as it is a pop music

CD, and the `collect()` operation places it in the list created to accumulate the results.

4. `cd3` is discarded by the `filter()` operation as it is not a pop music CD.
5. `cd4` is discarded by the `filter()` operation as it is not a pop music CD.

In Figure 16.1c, when the stream is exhausted, execution of the `collect()` terminal operation completes and execution of the pipeline stops. Note that there was only *one* pass over the elements in the stream. From Figure 16.1c, we see that the resulting list contains only `cd0` and `cd2`, which is the result of the query. Printing the resulting `popCDs` list produces the following output:

Click here to view code image

```
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>
```

A stream is considered *consumed* once a terminal operation has completed execution. A stream that has been consumed *cannot* be reused, and any attempt to use it will result in a nasty `java.lang.IllegalStateException`.

The code presented in this subsection is shown in Example 16.2.

Example 16.2 *Data Processing Using Streams*

Click here to view code image

```
import java.util.List;
import java.util.stream.Stream;

public class StreamPipeLine {
  public static void main(String[] args) {

    List<CD> cdList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd3, CD.cd4);

    // (A) Query to create a list of all CDs with pop music.
    List<CD> popCDs = cdList.stream()              // (1) Stream creation.
        .filter(CD::isPop)                         // (2) Intermediate operati
        .toList();                                 // (3) Terminal operation.
    System.out.println(popCDs);

    // (B) Equivalent to (A).
    Stream<CD> stream1 = cdList.stream();          // (1a) Stream creation.
    Stream<CD> stream2 = stream1.filter(CD::isPop);// (2a) Intermediate operat
    List<CD> popCDs2 = stream2.toList();           // (3a) Terminal operation.
    System.out.println(popCDs2);
  }
}
```

Output from the program:

Click here to view code image

```
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>
```

Comparing Collections and Streams

It is important to understand the distinction between collections and streams. Streams are not collections, and vice versa, but a stream can be created with a collection as the data source (p. 897).

Collections are data structures that can be used to store and retrieve elements. Streams are data structures that do not store their elements, but process them by expressing computations on them through operations like `filter()` and `collect()`.

Typically, operations are provided to add or remove elements from a collection. However, no elements can be added or removed from a stream—that is, streams are immutable. Because of their functional nature, if a stream operation does remove or discard an element in a stream, a new stream is returned with the remaining elements. A stream operation does not mutate its data source.

A collection can be used in the program as long as there is a reference to it. However, a stream cannot be reused once it is consumed. It must be re-created on the data source in order to be reused.

Operations on a collection are executed immediately, whereas streams can define intermediate operations that are executed on demand—that is, by lazy execution.

Collections are iterable, but streams are *not* iterable. Streams do not implement the `Iterable<T>` interface, and therefore, a `for(:)` loop cannot be used to iterate over a stream.

Mechanisms for iteration over a collection are based on an iterator defined by the `Collection` interface, but must be explicitly used in the program to iterate over the elements; this is called *external iteration*. On the other hand, iteration over stream elements is implicitly handled by the API; this is called *internal iteration* and it occurs when the stream operations are executed.

Collections have a finite size, but streams can be unbounded; these are called *infinite streams*. Special stream operations, such as `limit()`, exist to compute with infinite streams.

Some collections, such as lists, allow positional access of their elements with an index. However, this is not possible with streams, as only aggregate operations are permissible.

Note also that streams supported by the Stream API are not the same as those supported by the File I/O APIs (§20.1, p. 1233).

Overview of API for Data Processing Using Streams

In this subsection we present a brief overview of new interfaces and classes that are introduced in this chapter. We focus mainly on the Stream API in the `java.util.stream` package, but we also discuss utility classes from the `java.util` package.

The Stream Interfaces

Figure 16.2 shows the inheritance hierarchy of the core stream interfaces that are an important part of the Stream API defined in the `java.util.stream` package. The generic interface `Stream<T>` represents *a stream of object references*—that is, *object streams*. The interfaces `IntStream`, `LongStream`, and `DoubleStream` are specializations to *numeric streams* of type `int`, `long`, and `double`, respectively. These interfaces provide the static factory methods for creating streams from various sources (p. 890), and define the intermediate operations (p. 905) and the terminal operations (p. 946) on streams.

The interface `BaseStream` defines the basic functionality offered by all streams. It is recursively parameterized with a stream element type `T` and a subtype `S` of the `BaseStream` interface. For example, the `Stream<T>` interface is a subtype of the parameterized `BaseStream<T, Stream<T>>` interface, and the `IntStream` interface is a subtype of the parameterized `BaseStream<Integer, IntStream>` interface.

All streams implement the `AutoCloseable` interface, meaning they should be closed after use in order to facilitate resource management during execution. However, this is not necessary for the majority of streams. Only resource-backed streams need to be closed—for example, a stream whose data source is a file. Such resources are best managed automatically with the `try`-with-resources statement (§7.7, p. 407).

Figure 16.2 *The Core Stream Interfaces*

The `Collectors` Class

A collector encapsulates the machinery required to perform a reduction operation (p. 978). The `java.util.stream.Collector` interface defines the functionality that a collector must implement. The `java.util.stream.Collectors` class provides a rich set of predefined collectors for various kinds of reductions.

The Optional Classes

Instances of the `java.util.Optional<T>` class are containers that may or may not contain an object of type `T` (p. 940). An `Optional<T>` instance can be used to represent the absence of a value of type `T` more meaningfully than the `null` value. The numeric analogues are `OptionalInt`, `OptionalLong`, and `OptionalDouble` that can encapsulate an `int`, a `long`, or a `double` value, respectively.

The Numeric Summary Statistics Classes

Instances of the `IntSummaryStatistics`, `LongSummaryStatistics`, and `DoubleSummary-Statistics` classes in the `java.util` package are used by a group of reduction operations to collect summarizing statistics like the count, sum, average, min, and max of the values in a numeric stream of type `int`, `long`, and `double`, respectively (p. 974, p. 1001).

16.4 Building Streams

A stream must have a data source. In this section we will explore how streams can be created from various data sources: collections, arrays, specified values, generator functions, strings, and I/O channels, among others.

Aspects to Consider When Creating Streams

When creating a stream from a data source, certain aspects to consider include whether the stream is:

- Sequential or parallel
- Ordered or unordered
- Finite or infinite

- Object or numeric

Sequential or Parallel Stream

A *sequential stream* is one whose elements are processed sequentially (as in a `for` loop) when the stream pipeline is executed by a single thread. Figure 16.1 illustrates the execution of a sequential stream, where the stream pipeline is executed by a single thread.

A *parallel stream* is split into multiple substreams that are processed in parallel by multiple instances of the stream pipeline being executed by multiple threads, and their intermediate results combined to create the final result. Parallel streams are discussed in detail later (p. 1009).

The different ways to create a stream on a data source that are illustrated in this section result in a sequential stream. A parallel stream can only be created directly on a collection by invoking the `Collection.parallelStream()` method (p. 897).

The sequential or parallel mode of an existing stream can be modified by calling the `BaseStream.sequential()` and `BaseStream.parallel()` intermediate operations, respectively (p. 933). A stream is executed sequentially or in parallel depending on the execution mode of the stream on which the terminal operation is initiated.

Ordered or Unordered Stream

The *encounter order* of a stream refers to the way in which a stream makes its elements available for processing to an operation in a pipeline. For such data sources as a list, the encounter order of the initial stream is the same as the order of the elements in the list, whereas a stream created with a set of values does not have an encounter order, as the elements of a set are considered to be unordered.

The encounter order of a stream may be changed by an intermediate operation. For example, the `sorted()` operation may impose an encounter order on an unordered stream (p. 929), and the `unordered()` operation may designate a stream unordered (p. 932). Also, some terminal operations might choose to ignore the encounter order; an example is the `forEach()` operation (p. 948).

For ordered sequential streams, an identical result is produced when identical stream pipelines are executed on an identical data source—that is, the execution is deterministic. This guarantee does not hold for unordered sequential streams, as the results produced might be different.

Processing of unordered parallel streams may have better performance than for ordered parallel streams in identical stream pipelines when the ordering constraint is removed, as maintaining the order might carry a performance penalty.

Finite or Infinite Stream

The size of a stream can be finite or infinite depending on how the stream is created. The `generate()` and `iterate()` methods of the core stream interfaces create streams with an infinite number of elements (p. 894). Such a stream is said to be *unbounded*. The overloaded `ints()`, `longs()`, and `doubles()` methods of the `java.util.Random` class create streams with an effectively unlimited number of pseudorandom values (p. 900). An infinite stream must be truncated before the terminal operation is initiated; otherwise, the stream pipeline will never terminate (p. 917).

Object or Numeric Stream

The interface `Stream<T>` defines the contract for streams of object references—that is, *object streams*. The specialized interfaces `IntStream`, `LongStream`, and `DoubleStream` represent streams of `int`, `long`, and `double` values, respectively—that is, *numeric streams*. The various ways to create streams discussed here will always result in a stream whose element type is either a reference type or a numeric type (`int`, `long`, or `double`). Conversion between these stream types is discussed in §16.5, p. 934.

Table 16.1, p. 904, summarizes selected methods for building streams from various data sources.

The following `static` factory methods for building streams are defined in the `Stream<T>` class. Counterparts to these methods are also provided by the `IntStream`, `LongStream`, and `DoubleStream` interfaces for creating numeric streams, unless otherwise noted:

> Click here to view code image
>
> ```
> static <T> Stream<T> empty()
> static <T> Stream<T> of(T t)
> static <T> Stream<T> of(T... varargs)
> ```
>
> The first method creates an *empty* stream—that is, a stream

that has no elements.

The second method creates a *singleton* stream—that is, a stream that has a single element.

The third method creates a *finite sequential ordered* stream whose elements are the values specified by the variable arity parameter `varargs`.

The second and last methods throw a `NullPointerException` if the argument is `null`.

Click here to view code image

```
static <T> Stream<T> ofNullable(T t)
```

Only in the `Stream<T> interface.`

Creates a *singleton* stream that has a single element, if the argument `t` is non-null; otherwise, it returns an empty stream.

Click here to view code image

```
static <T> Stream<T> generate(Supplier<T> supplier)
```

Creates an *infinite sequential unordered* stream where each element is generated by the specified `supplier`. Typically, this is used for constant streams and streams with random elements.

Click here to view code image

```
static <T> Stream<T> iterate(T s, UnaryOperator<T> uop)
```

Creates an *infinite sequential ordered* stream produced by the iterative application of the function `uop` to an initial element seed `s`, producing a stream consisting of `s`, `e1=uop.apply(s)`, `e2=uop.apply(e1)`, and so on; that is, `uop` is applied to the previous element.

Click here to view code image

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

Creates a stream whose elements are all elements of the first stream followed by all elements of the second stream.

The resulting stream is ordered only if both input streams are ordered. If either input stream is parallel, the resulting stream is parallel. As one would expect, the resulting stream is finite only if both input streams are finite.

The Empty Stream

An *empty stream* can be obtained by calling the `empty()` method of the core stream interfaces. As the name implies, such a stream has no elements.

Click here to view code image

```
Stream<CD> cdStream = Stream.empty();                    // Empty stream of CD.
System.out.println("Count: " + cdStream.count());        // Count: 0
IntStream iStream = IntStream.empty();                   // Empty stream of int
System.out.println("Count: " + iStream.count());         // Count: 0
```

The `count()` method is a terminal operation in the `Stream<T>` interface (p. 953). It returns the number of elements processed through the stream pipeline.

Using the `null` value to indicate that a stream is empty may result in a `NullPointer-Exception`. Therefore, using an explicit empty stream is highly recommended.

Streams from Specified Values

The two overloaded `of()` methods in the core stream interfaces create finite sequential ordered streams from data values that are specified as arguments.

In the code below, the single-argument `of()` method is called at (1), and the variable arity `of()` method is called at (2), both creating a stream of element type `CD`. The size of the streams created at (1) and (2) is 1 and 3, respectively. The stream pipeline comprising (3) and (4) filters the pop music CDs and prints their title at (4). The `forEach()` terminal operation at (4) applies its `Consumer` action to each pop music CD.

Click here to view code image

```
// From specified objects.
Stream<CD> cdStream1 = Stream.of(CD.cd0);                 // (1) Single-arg c
Stream<CD> cdStream2 = Stream.of(CD.cd0, CD.cd1, CD.cd2); // (2) Varargs call
cdStream2.filter(CD::isPop)                               // (3)
         .forEach(cd -> System.out.println(cd.title()));  // (4)
```

The code below shows examples of using numeric values to create streams. The values specified at (1) and (2) are autoboxed to create a stream of objects. The declaration statements at (3) and (4) avoid the overhead of autoboxing when streams of numeric values are created. However, at (4), an implicit numeric conversion to `double` is applied to the non-`double` values.

Click here to view code image

```
// From specified numeric values.
Stream<Integer> integerStream1 = Stream.of(2017, 2018, 2019);       // (1)
Stream<? extends Number> numStream = Stream.of(100, 3.14D, 5050L);   // (2)
IntStream intStream1 = IntStream.of(2017, 2018, 2019);              // (3)
DoubleStream doubleStream = DoubleStream.of(100, 3.14D, 5050L);     // (4)
```

The variable arity `of()` method can be used to create a stream whose source is an array. Equivalently, the overloaded `Arrays.stream()` method can be used for the same purpose. In all cases below, the size of the stream is the same as the size of the array, except at (7). An `int` array is an object that is passed to the single-argument `Stream.of()` method (creating a `Stream<int[]>`), and not the variable arity `Stream.of()` method. The `int` array is, however, passed to the variable arity `IntStream.of()` method at (8). Creating a stream from a numeric array is safer with the numeric stream interfaces or the `Arrays.stream()` method than the `Stream.of()` method.

Click here to view code image

```
// From an array of CDs.
Stream<CD> cdStream3 = Stream.of(CD.cdArray);                    // (1)
Stream<CD> cdStream4 = Arrays.stream(CD.cdArray);                // (2)

// From an array of Integer.
Integer[] integerArray = {2017, 2018, 2019};                     // (3)
Stream<Integer> integerStream2 = Stream.of(integerArray);        // (4)
Stream<Integer> integerStream3 = Arrays.stream(integerArray); // (5)

// From an array of int.
int[] intArray = {2017, 2018, 2019};                             // (6)
Stream<int[]> intArrayStream = Stream.of(intArray);              // (7) Size is 1
IntStream intStream2 = IntStream.of(intArray);                   // (8) Size is 3
IntStream intStream3 = Arrays.stream(intArray);                  // (9) Size is 3
```

The `Stream.of()` methods throw a `NullPointerException` if the argument is `null`. The `ofNullable()` method, on the other hand, returns an empty stream if this is the case; otherwise, it returns a singleton stream.

Using Generator Functions to Build Infinite Streams

The `generate()` and `iterate()` methods of the core stream interfaces can be used to create infinite sequential streams that are unordered or ordered, respectively.

Infinite streams need to be truncated explicitly in order for the terminal operation to complete execution, or the operation will not terminate. Some stateful intermediate operations must process all elements of the streams in order to produce their results—for example, the `sort()` intermediate operation (p. 929) and the `reduce()` terminal operation (p. 955). The `limit(maxSize)` intermediate operation can be used to limit the number of elements that are available for processing from a stream (p. 917).

Generate

The `generate()` method accepts a *supplier* that generates the elements of the infinite stream.

Click here to view code image

```
IntSupplier supplier = () -> (int) (6.0 * Math.random()) + 1;  // (1)
IntStream diceStream = IntStream.generate(supplier);          // (2)
diceStream.limit(5)                                            // (3)
          .forEach(i -> System.out.print(i + " "));           // (4) 2 4 5 2
```

The `IntSupplier` at (1) generates a number between 1 and 6 to simulate a dice throw every time it is executed. The supplier is passed to the `generate()` method at (2) to create an infinite unordered `IntStream` whose values simulate throwing a dice. In the pipeline comprising (3) and (4), the number of values in the `IntStream` is limited to 5 at (3) by the `limit()` intermediate operation, and the value of each dice throw is printed by the `forEach()` terminal operation at (4). We can expect five values between 1 and 6 to be printed when the pipeline is executed.

Iterate

The `iterate()` method accepts a *seed* value and a *unary operator*. The method generates the elements of the infinite ordered stream *iteratively*: It applies the operator to the previous element to generate the next element, where the first element is the seed value.

In the code below, the seed value of 1 is passed to the `iterate()` method at (2), together with the unary operator `uop` defined at (1) that increments its argument by 2. The first element is 1 and the second element is the result of the unary operator applied to 1, and so on. The `limit()` operation limits the stream to five values. We can expect the `forEach()` operation to print the first five odd numbers.

Click here to view code image

```
IntUnaryOperator uop = n -> n + 2;                // (1)
IntStream oddNums = IntStream.iterate(1, uop);    // (2)
oddNums.limit(5)
```

```
       .forEach(i -> System.out.print(i + " "));  // 1 3 5 7 9
```

The following stream pipeline will really go bananas if the stream is not truncated by the `limit()` operation:

Click here to view code image

```
Stream.iterate("ba", b -> b + "na")
      .limit(5)
      .forEach(System.out::println);
```

Concatenating Streams

The `concat()` method creates a resulting stream where the elements from the first argument stream are followed by the elements from the second argument stream. The code below illustrates this operation for two unordered sequential streams. Two sets are created at (1) and (2) based on lists of strings that are passed to the set constructors. The two streams created at (3) and (4) are unordered, since they are created from sets (p. 897). These unordered streams are passed to the `concat()` method at (5). The resulting stream is processed in the pipeline comprising (5) and (6). The `forEachOrdered()` operation at (6) respects the encounter order of the stream if it has one—that is, if it is ordered (p. 948). The output confirms that the resulting stream is unordered.

Click here to view code image

```
Set<String> strSet1                                                         //
    = Set.of("All", " objects", " are", " equal");
Set<String> strSet2                                                         //
    = Set.of(" but", " some", " are", " more", " equal", " than", " others.");
Stream<String> unorderedStream1 = strSet1.stream();                         //
Stream<String> unorderedStream2 = strSet2.stream();                         //
Stream.concat(unorderedStream1, unorderedStream2)                           //
      .forEachOrdered(System.out::print);
// objectsAll equal are some are others. than equal more but
```

The resulting stream is ordered if both argument streams are ordered. The code below illustrates this operation for two ordered sequential streams. The two streams created at (1) and (2) below are ordered. The ordering is given by the specification order of the strings as arguments to the `Stream.of()` method. These ordered streams are passed to the `concat()` method at (3). The resulting stream is processed in the pipeline comprising (3) and (4). The output confirms that the resulting stream is ordered.

Click here to view code image

```
Stream<String> orderedStream1 = Stream.of("All", " objects",                //
                                          " are", " equal");
Stream<String> orderedStream2 = Stream.of(" but", " some", " are", " more", //
                                          " equal", " than", " others.");
Stream.concat(orderedStream1, orderedStream2)                               //
      .forEachOrdered(System.out::print);                                   //
// All objects are equal but some are more equal than others.
```

As far as the mode of the resulting stream is concerned, it is parallel if at least one of the constituent streams is parallel. The code below illustrates this behavior.

The `parallel()` intermediate operation used at (1) returns a possibly parallel stream (p. 933). The call to the `isParallel()` method confirms this at (2). We pass one parallel stream and one sequential stream to the `concat()` method at (3). The call to the `isParallel()` method at (4) confirms that the resulting stream is parallel. The printout from (5) shows that it is also unordered. Note that new streams are created on the sets `strSet1` and `strSet2` at (1) and (3), respectively, as we cannot reuse the streams that were created earlier and consumed.

Click here to view code image

```
Stream<String> pStream1 = strSet1.stream().parallel();                // (1)
System.out.println("pStream1 is parallel: " + pStream1.isParallel()); // (2) t
Stream<String> rStream = Stream.concat(pStream1, strSet2.stream());   // (3)
System.out.println("rStream is parallel: " + pStream1.isParallel());  // (4) t
rStream.forEachOrdered(System.out::print);                            // (5)
// objectsAll equal are some are others. than equal more but
```

Streams from Collections

The `default` methods `stream()` and `parallelStream()` of the `Collection` interface create streams with collections as the data source. Collections are the only data source that provide the `parallelStream()` method to create a parallel stream directly. Otherwise, the `parallel()` intermediate operation must be used in the stream pipeline.

The following `default` methods for building streams from collections are defined in the `java.util.Collection` interface:

> Click here to view code image
>
> ```
> default Stream<E> stream()
> default Stream<E> parallelStream()
> ```
>
> Return a *finite sequential* stream or a possibly *parallel* stream with this collection as its source, respectively. Whether it is ordered or not depends on the collection used as the data

> source.

We have already seen examples of creating streams from lists and sets, and several more examples can be found in the subsequent sections.

The code below illustrates two points about streams and their data sources. If the data source is modified before the terminal operation is initiated, the changes will be reflected in the stream. A stream is created at (2) with a list of CDs as the data source. Before a terminal operation is initiated on this stream at (4), an element is added to the underlying data source list at (3). Note that the list created at (1) is modifiable. The `count()` operation correctly reports the number of elements processed in the stream pipeline.

Click here to view code image

```
List<CD> listOfCDS = new ArrayList<>(List.of(CD.cd0, CD.cd1));       // (1)
Stream<CD> cdStream = listOfCDS.stream();                            // (2)
listOfCDS.add(CD.cd2);                                               // (3)
System.out.println(cdStream.count());                                // (4) 3
// System.out.println(cdStream.count());                // (5) IllegalStateExcept
```

Trying to initiate an operation on a stream whose elements have already been consumed results in a `java.lang.IllegalStateException`. This case is illustrated at (5). The elements in the `cdStream` were consumed after the terminal operation at (4). A new stream must be created on the data source before any stream operations can be run.

To create a stream on the entries in a `Map`, a collection view can be used. In the code below, a `Map` is created at (1) and populated with some entries. An entry view on the map is obtained at (2) and used as a data source at (3) to create an unordered sequential stream. The terminal operation at (4) returns the number of entries in the map.

Click here to view code image

```
Map<Integer, String> dataMap = new HashMap<>();                     // (1)
dataMap.put(1, "en"); dataMap.put(2, "to");
dataMap.put(3, "tre"); dataMap.put(4, "fire");
long numOfEntries = dataMap
    .entrySet()                                                     // (2)

    .stream()                                                       // (3)
    .count();                                                       // (4) 4
```

In the examples in this subsection, the call to the `stream()` method can be replaced by a call to the `parallelStream()` method. The stream will then execute in parallel, without the need for any additional synchronization code (p. 1009).

Streams from Arrays

We have seen examples of creating streams from arrays when discussing the variable arity `of()` method of the stream interfaces and the overloaded `Arrays.stream()` methods earlier in the chapter (p. 893). The sequential stream created from an array has the same order as the positional order of the elements in the array. As far as numeric streams are concerned, only an `int`, `long`, or `double` array can act as the data source of such a stream.

The code below illustrates creating a stream based on a subarray that is given by the half-open interval specified as an argument to the `Array.stream()` method, as shown at (1). The stream pipeline at (2) calculates the length of the subarray.

Click here to view code image

```
Stream<CD> cdStream = Arrays.stream(cdArray, 1, 4);           // (1)
long noOfElements = cdStream.count();                          // (2) 3
```

The following overloaded `static` methods for building sequential ordered streams from arrays are defined in the `java.util.Arrays` class:

> Click here to view code image
>
> ```
> static <T> Stream<T> stream(T[] array)
> static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)
> ```
>
> Create a *finite sequential ordered* `Stream<T>` with the specified array as its source. The stream created by the second method comprises the range of values given by the specified half-open interval.
>
> Click here to view code image
>
> ```
> static NumTypeStream stream(numtype[] array)
> static NumTypeStream stream(numtype[] array,
>                             int startInclusive, int endExclusive)
> ```
>
> *NumType* is `Int`, `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`.
>
> Create a *finite sequential ordered NumType*`Stream` (which is either `IntStream`, `LongStream`, or `DoubleStream`) with the specified array as its source. The stream created by the second method comprises the range of primitive values given by the

> specified half-open interval.

Building a Numeric Stream with a Range

The overloaded methods `range()` and `rangeClosed()` can be used to create *finite ordered* streams of integer values based on a range that can be *half-open* or *closed*, respectively. The increment size is always 1.

The following static factory methods for building numeric streams are defined only in the `IntStream` and `LongStream` interfaces in the `java.util.stream` package.

> Click here to view code image
>
> ```
> static NumTypeStream range(numtype startInclusive, numtype endExclusive)
> static NumTypeStream rangeClosed(numtype startInclusive,
>                                  numtype endInclusive)
> ```
>
> *NumType* is `Int` or `Long`, and the corresponding *`numtype`* is `int` or `long`.
>
> Both methods return a *finite sequential ordered NumType*`Stream` whose elements are a sequence of numbers, where the first number in the stream is the start value of the range `startInclusive` and increment length of the sequence is 1. For a *half-open interval*, as in the first method, the end value of the range `endExclusive` is excluded. For a *closed interval*, as in the second method, the end value of the range `endInclusive` is included.

The `range(startInclusive, endExclusive)` method is equivalent to the following `for(;;)` loop:

Click here to view code image

```
for (int i = startInclusive; i < endExclusive; i++) {
  // Loop body.
}
```

When processing with ranges of integer values, the `range()` methods should also be considered on par with the `for(;;)` loop.

The stream pipeline below prints all the elements in the `CD` array in reverse. Note that no terminating condition or increment expression is specified. As range values are always in increasing order, a simple adjustment can be done to reverse their order.

Click here to view code image

```
IntStream.range(0, CD.cdArray.length)                                     // (1)
         .forEach(i -> System.out.println(cdArray[CD.cdArray.length - 1 - i]))
```

The following example counts the numbers that are divisible by a specified divisor in a given range of values.

Click here to view code image

```
int divisor = 5;
int start = 2000, end = 3000;
long divisibles = IntStream
    .rangeClosed(start, end)                                              // (1)
    .filter(number -> number % divisor == 0)                              // (2)
    .count();                                                             // (3)
System.out.println(divisibles);                                           // 201
```

The next example creates an `int` array that is filled with increment values specified by the range at (1) below. The `toArray()` method is a terminal operation that creates an array of the appropriate type and populates it with the values in the stream (p. 971).

Click here to view code image

```
int first = 10, len = 8;
int[] intArray = IntStream.range(first, first + len).toArray();           // (1)
System.out.println(intArray.length + ": " + Arrays.toString(intArray));
//8: [10, 11, 12, 13, 14, 15, 16, 17]
```

The example below shows usage of two nested ranges to print the multiplication tables. The inner arrange is executed 10 times for each value in the outer range.

Click here to view code image

```
IntStream.rangeClosed(1, 10)                                   // Outer range.
         .forEach(i -> IntStream.rangeClosed(1, 10)            // Inner range.
                         .forEach(j -> System.out.printf("%2d * %2d = %2d%n"
                                                      i, j, i * j)));
}
```

We cordially invite the inquisitive reader to code the above examples in the imperative style using explicit loops. Which way is better is not always that clear-cut.

Numeric Streams Using the `Random` Class

The following methods for building *numeric unordered* streams are defined in the `java.util.Random` class:

> *NumType* is `Int`, `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`. The corresponding

> overloaded `numtypes()` methods are `ints()`, `longs()`, and `doubles()`.
>
> Click here to view code image
>
> ```
> NumTypeStream numtypes()
> NumTypeStream numtypes(numtype randomNumberOrigin,
>                        numtype randomNumberBound)
> NumTypeStream numtypes(long streamSize)
> NumTypeStream numtypes(long streamSize, numtype randomNumberOrigin,
>                        numtype randomNumberBound)
> ```
>
> The first two methods generate an *effectively unlimited sequential unordered* stream of pseudorandom `numtype` values. For the zero-argument `doubles()` method, the `double` values are between 0.0 (inclusive) and 1.0 (exclusive). For the second method, the `numtype` values generated are in the *half-open* interval defined by the origin and the bound values.
>
> The last two methods generate a *finite sequential unordered* stream of pseudorandom `numtype` values, where the length of the stream is limited by the specified `streamSize` parameter.

The examples below illustrate using a *pseudorandom number generator* (PRNG) to create numeric streams. The same PRNG can be used to create multiple streams. The PRNG created at (1) will be used in the examples below.

Click here to view code image

```
Random rng = new Random();                              // (1)
```

The `int` stream created at (2) is an *effectively unlimited unordered* stream of `int` values. The size of the stream is limited to 3 by the `limit()` operation. However, at (3), the maximum size of the stream is specified in the argument to the `ints()` method. The values in both streams at (2) and (3) can be any random `int` values. The contents of the array constructed in the examples will, of course, vary.

Click here to view code image

```
IntStream iStream = rng.ints();                         // (2) Unlimited, any int va
int[] intArray = iStream.limit(3).toArray();            // Limits size to 3
//[-1170441471, 1070948914, 264046613]
intArray = rng.ints(3).toArray();                       // (3) Size 3, any int value
//[1011448344, -974832344, 816809715]
```

The unlimited unordered stream created at (4) simulates the dice throw we implemented earlier using the `generate()` method (p. 895). The values are between 1 and 6, inclusive. The `limit()`

method must be used explicitly to limit the stream. The finite unordered stream created at (5) incorporates the size and the value range.

Click here to view code image

```
intArray = rng.ints(1, 7)                         // (4) Unlimited, [1, 6]
              .limit(3)                           // Limits size to 3
              .toArray();                         // [5, 2, 4]

intArray = rng.ints(3, 1, 7)                      // (5) Size 3, [1, 6]
              .toArray();                         // [1, 4, 6]
```

The zero-argument `doubles()` method and the single-argument `doubles(streamSize)` method generate an unlimited and a limited unordered stream, respectively, whose values are between 0.0 and 1.0 (exclusive).

Click here to view code image

```
DoubleStream dStream = rng.doubles(3);            // (6) Size 3, [0.0, 1.0)
double[] dArray = dStream.toArray();
//[0.9333794789872794, 0.7037326827186609, 0.2839257522887708]
```

Streams from a `CharSequence`

The `CharSequence.chars()` method creates a *finite sequential ordered* `IntStream` from a sequence of `char` values. The `IntStream` must be transformed to a `Stream<Character>` in order to handle the values as `Character`s. The `IntStream.mapToObj()` method can be used for this purpose, as shown at (2). A cast is necessary at (2) in order to convert an `int` value to a `char` value which is autoboxed in a `Character`. Conversion between streams is discussed in §16.5, p. 934.

Click here to view code image

```
String strSource = "banananana";
IntStream iStream = strSource.chars();                   // (1)
iStream.forEach(i -> System.out.print(i + " "));         // Prints ints.
// 98 97 110 97 110 97 110 97 110 97

strSource.chars()
         .mapToObj(i -> (char)i)                         // (2) Stream<Charact
         .forEach(System.out::print);                    // Prints chars.
// banananana
```

The following default method for building `IntStream`s from a sequence of `char` values (e.g., `String` and `StringBuilder`) is defined in the `java.lang.CharSequence` interface (§8.4, p. 444):

```
default IntStream chars()
```

Creates a *finite sequential ordered* stream of `int` values by zero-extending the `char` values in this sequence.

Streams from a `String`

The following method of the `String` class can be used to extract *text lines* from a string:

```
Stream<String> lines()
```

Returns a stream of lines extracted from this string, separated by line terminators.

In the code below, the string at (1) contains three text lines separated by the line terminator (`\n`). A stream of element type `String` is created using this string as the source at (2). Each line containing the word `"mistakes"` in this stream is printed at (3).

Click here to view code image

```
String inputLines = "Wise men learn from their mistakes.\n"                 //
                  + "But wiser men learn from the mistakes of others.\n"
                  + "And fools just carry on.";
Stream<String> lStream = inputLines.lines();                                //
lStream.filter(l -> l.contains("mistakes")).forEach(System.out::println);   //
```

Output from the code:

Click here to view code image

```
Wise men learn from their mistakes.
But wiser men learn from the mistakes of others.
```

Streams from a `BufferedReader`

A `BufferedReader` allows contents of a text file to be read as lines. A *line* is a sequence of characters terminated by a *line terminator sequence*. Details of using a `Buffered-Reader` are covered in §20.3, p. 1251. A simple example of creating streams on text files using a `BufferedReader` is presented below.

At (1) and (2) in the header of the `try`-with-resources statement (§7.7, p. 407), a `BufferedReader` is created to read lines from a given file, and a stream of `String` is created at (3) by the `lines()` method provided by the `BufferedReader` class. These declarations are permissible since both the buffered reader and the stream are

`Auto-Closeable`. Both will be automatically closed after the `try` block completes execution. A terminal operation is initiated at (4) on this stream to count the number of lines in the file. Of course, each line from the stream can be processed depending on the problem at hand.

Click here to view code image

```
try ( FileReader fReader = new FileReader("CD_Data.txt");           // (1)
      BufferedReader bReader = new BufferedReader(fReader);         // (2)
      Stream<String> lStream = bReader.lines() ) {                  // (3)
  System.out.println("Number of lines: " + lStream.count());       // (4) 13
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

The following method for building a `Stream<String>` from a text file is defined in the `java.io.BufferedReader` class:

> `Stream<String> lines()`
>
> Returns a *finite sequential ordered* `Stream` of `String`s, where the elements are text lines read by this `BufferedReader`.
>
> The reader position in the file is not guaranteed after the stream terminal operation completes.
>
> The result of the terminal stream operation is undefined if the reader is operated upon during the execution of this operation.
>
> Any operation on a `Stream` returned by a `BufferedReader` that has already been closed will throw an `UncheckedIOException`.

Streams from Factory Methods in the `Files` Class

A detailed discussion of the NIO2 File API that provides the classes for creating the various streams for reading files, finding files, and walking directories in the file system can be found in Chapter 21, p. 1285.

Analogous to the `lines()` method in the `BufferedReader` class, a `static` method with the same name is provided by the `java.nio.file.Files` class that creates a stream for reading the file content as lines.

In the example below, a `Path` is created at (1) to represent a file on the file system. A stream is created to read lines from the path at

(2) in the header of the `try`-with-resources statement (§7.7, p. 407). As streams are `AutoCloseable`, such a stream is automatically closed after the `try` block completes execution. As no character set is specified, bytes from the file are decoded into characters using the UTF-8 charset. A terminal operation is initiated at (3) on this stream to count the number of lines in the file. Again, each line in the stream can be processed as desired.

Click here to view code image

```
Path path = Paths.get("CD_Data.txt");                                  // (1)
try (Stream<String> lStream = Files.lines(path)) {                     // (2)
  System.out.println("Number of lines: " + lStream.count());          // (3) 13
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

The following static methods for building a `Stream<String>` from a text file are defined in the `java.nio.file.Files` class:

> Click here to view code image
>
> ```
> static Stream<String> lines(Path path)
> static Stream<String> lines(Path path, Charset cs)
> ```
>
> Return a *finite sequential ordered* `Stream` of `String`, where the elements are text lines read from a file given by the specified `path`. The first method decodes the bytes into characters using the UTF-8 charset. The charset to use can be explicitly specified, as in the second method.

Summary of Stream Building Methods

Selected methods for building streams from various data sources are listed in Table 16.1. The first column lists the method names and the reference type that provides them. For brevity, the parameters of the methods are omitted. Note that some methods are overloaded. The prefix *`NumType`* stands for `Int`, `Long`, or `Double`. A reference is also provided where details about the method can be found. The remaining four columns indicate various aspects of a stream: the type of stream returned by a method, whether the stream is finite or infinite, whether it is sequential or parallel, and whether it is ordered or unordered (p. 891).

Table 16.1 *Summary of Stream Building Methods*

Method	Stream type	Finite/Infinite	Sequential/Parallel	Ordered/Unordered
`Stream.empty()`	`Stream<T>`			

`Stream.empty()` (p. 893)	*Ordered*	*Finite*	*Sequential*

`Stream.of()`
`Stream.ofNullable()`
`NumTypeStream.of()` (p. 893)
	Ordered	*Finite*	*Sequential*

`Stream.generate()` `NumTypeStream.generate()` (p. 895)	*Unordered*	*Infinite*	*Sequential*
`Stream.iterate()` `NumTypeStream.iterate` (p. 895)	*Ordered*	*Infinite*	*Sequential*
`Stream.concat()` `NumTypeStream.concat()` (p. 895)	*Ordered if both ordered*	*Finite if both finite*	*Parallel if either parallel*
`Collection.stream()` (p. 897)	*Ordered if collection ordered*	*Finite*	*Sequential*
`Collection.parallelStream()` (p. 897)	*Ordered if collection ordered*	*Finite*	*Parallel*
`Arrays.stream()` `NumTypeStream` (p. 898)	*Ordered*	*Finite*	*Sequential*

`IntStream.range()`
`IntStream.rangeClosed()`
`LongStream.range()`
`LongStream.rangeClosed()` (p. 898)
	Ordered	*Finite*	*Sequential*

`Random.ints()`
`Random.longs()`
`Random.doubles()` (p. 900)
	Unordered	*Finite or infinite, depending on parameters*	*Sequential*

`CharSequence.chars()` (p. 901)	*Ordered*	*Finite*	*Sequential*
`String.lines()` (p. 902)	*Ordered*	*Finite*	*Sequential*
`BufferedReader.lines()` (p. 902)	*Ordered*	*Finite*	*Sequential*
`Files.lines()` (p. 903)	*Ordered*	*Finite*	*Sequential*

16.5 Intermediate Stream Operations

A stream pipeline is composed of stream operations. The stream operations process the elements of the stream to produce some result. After the creation of the initial stream, the elements of the stream are processed by zero or more intermediate operations before the mandatory terminal operation reduces the elements to some final result. The initial stream can undergo several *stream transformations* (technically called *mappings*) by the intermediate operations as the elements are processed through the pipeline.

Intermediate operations *map* a stream to a new stream, and the terminal operation *reduces* the final stream to some result. Because of the nature of the task they perform, the operations in a stream pipeline are also called *map-reduce transformations*.

Aspects of Streams, Revisited

We now take a closer look at the following aspects pertaining to streams:

- Stream mapping
- Lazy execution
- Short-circuit evaluation
- Stateless and stateful operations
- Order of intermediate operations
- Non-interfering and stateless behavioral parameters of stream operations

Table 16.3, p. 938, summarizes certain aspects of each intermediate operation. Table 16.4, p. 939, summarizes the intermediate operations provided by the Stream API.

Stream Mapping

Each intermediate operation returns a new stream—that is, it maps the elements of its input stream to an output stream. Intermediate operations can thus be easily recognized. Having a clear idea of the *type* of the new stream an intermediate operation should produce aids in customizing the operation with an appropriate implementation of its behavioral parameters. Typically, these behavioral parameters are functional interfaces.

Because intermediate operations return a new stream, calls to methods of intermediate operations can be *chained*, so much so that code written in this *method chaining* style has become a distinct hallmark of expressing queries with streams.

In Example 16.3, the stream pipeline represents the query to create a list with titles of pop music CDs in a given list of CDs. Stream mapping is illustrated at (1). The initial stream of CDs (`Stream<CD>`) is first transformed by an intermediate operation (`filter()`) to yield a new stream that has only pop music CDs (`Stream<CD>`), and this stream is then transformed to a stream of CD titles

(`Stream<String>`) by a second intermediate operation (`map()`, p. 921). The stream of CD titles is reduced to the desired result (`List<CD>`) by the terminal operation (`collect()`).

In summary, the *type of the output stream* returned by an intermediate operation need not be the same as the *type of its input stream.*

Example 16.3 *Stream Mapping and Loop Fusion*

Click here to view code image

```
 import java.util.List;

public class StreamOps {
  public static void main(String[] args) {

    // Query: Create a list with titles of pop music CDs.

    // (1) Stream Mapping:
    List<CD> cdList1 = CD.cdList;
    List<String> popCDs1 = cdList1
        .stream()                      // Initial stream:         Stream<CD>
        .filter(CD::isPop)             // Intermediate operation: Stream<CD>
        .map(CD::title)                // Intermediate operation: Stream<Strin
        .toList();                     // Terminal operation: List<String>
    System.out.println("Pop music CDs: " + popCDs1);
    System.out.println();

    // (2) Lazy Evaluation:
    List<CD> cdList2 = CD.cdList;
    List<String> popCDs2 = cdList2
        .stream()                      // Initial stream:          Stream<CD>
        .filter(cd -> {                // Intermediate operation:  Stream<CD>
           System.out.println("Filtering: " + cd                     // (3)
                              + (cd.isPop() ? " is pop CD." : " is not pop CD.
           return cd.isPop();
         })
        .map(cd -> {                   // Intermediate operation: Stream<Strin
           System.out.println("Mapping: " + cd.title());          // (4)
           return cd.title();
         })
        .toList();                     // Terminal operation: List<String>
    System.out.println("Pop music CDs: " + popCDs2);
  }
}
```

Output from the program:

Click here to view code image

```
Pop music CDs: [Java Jive, Lambda Dancing]

Filtering: <Jaav, "Java Jive", 8, 2017, POP> is pop CD.
Mapping: Java Jive
Filtering: <Jaav, "Java Jam", 6, 2017, JAZZ> is not pop CD.
Filtering: <Funkies, "Lambda Dancing", 10, 2018, POP> is pop CD.
Mapping: Lambda Dancing
Filtering: <Genericos, "Keep on Erasing", 8, 2018, JAZZ> is not pop CD.
```

```
Filtering: <Genericos, "Hot Generics", 10, 2018, JAZZ> is not pop CD.
Pop music CDs: [Java Jive, Lambda Dancing]
```

Lazy Execution

A stream pipeline does not execute until a terminal operation is invoked. In other words, its intermediate operations do not start processing until their results are needed by the terminal operation. Intermediate operations are thus *lazy*, in contrast to the terminal operation, which is *eager* and executes when it is invoked.

An intermediate operation is *not* performed on all elements of the stream before performing the next operation on all elements resulting from the previous stream. Rather, the intermediate operations are performed back-to-back on *each* element in the stream. In a sense, the loops necessary to perform each intermediate operation on *all* elements successively are fused into a single loop (technically called *loop fusion*). Thus only a single pass is required over the elements of the stream.

Example 16.3 illustrates loop fusion resulting from lazy execution of a stream pipeline at (2). The intermediate operations now include print statements to announce their actions at (3) and (4). Note that we do not advocate this practice for production code. The output shows that the elements are processed *one at a time* through the pipeline when the terminal operation is executed. A CD is filtered first, and if it is a pop music CD, it is mapped to its title and the terminal operation includes this title in the result list. Otherwise, the CD is discarded. When there are no more CDs in the stream, the terminal operation completes, and the stream is consumed.

Short-circuit Evaluation

The lazy nature of streams allows certain kinds of optimizations to be performed on stream operations. We have already seen an example of such an optimization that results in loop fusion of intermediate operations.

In some cases, it is not necessary to process all elements of the stream in order to produce a result (technically called *short-circuit execution*). For instance, the `limit()` intermediate operation creates a stream of a specified size, making it unnecessary to process the rest of the stream once this limit is reached. A typical example of its usage is to turn an infinite stream into a finite stream. Another example is the `takeWhile()` intermediate operation that short-circuits stream processing once its predicate becomes `false`.

Certain terminal operations (`anyMatch()`, `allMatch()`, `noneMatch()`, `findFirst()`, `findAny()`) are also short-circuit operations, since they do not need to process all elements of the stream in order to produce a result (p. 949).

Stateless and Stateful Operations

An *stateless* operation is one that can be performed on a stream element without taking into consideration the outcome of any processing done on previous elements or on any elements yet to be processed. In other words, the operation does not retain any *state* from processing of previous elements in order to process a new element. Rather, the operation can be performed on an element independently of how the other elements are processed.

A *stateful* operation is one that needs to retain state from previously processed elements in order to process a new element.

The intermediate operations `distinct()`, `dropWhile()`, `limit()`, `skip()`, `sorted()`, and `takeWhile()` are *stateful* operations. All other intermediate operations are *stateless*. Examples of stateless intermediate operations include the `filter()` and `map()` operations.

Order of Intermediate Operations

The order of intermediate operations in a stream pipeline can impact the performance of a stream pipeline. If intermediate operations that reduce the size of the stream can be performed earlier in the pipeline, fewer elements need to be processed by the subsequent operations.

Moving intermediate operations such as `filter()`, `distinct()`, `dropWhile()`, `limit()`, `skip()`, and `takeWhile()` earlier in the pipeline can be beneficial, as they all decrease the size of the input stream. Example 16.4 implements two stream pipelines at (1) and (2) to create a list of CD titles, but skipping the first three CDs. The `map()` operation transforms each CD to its title, resulting in an output stream with element type `String`. The example shows how the number of elements processed by the `map()` operation can be reduced if the `skip()` operation is performed before the `map()` operation (p. 921).

Example 16.4 *Order of Intermediate Operations*

Click here to view code image

```
import java.util.List;

public final class OrderOfOperations {
  public static void main(String[] args) {

    List<CD> cdList = CD.cdList;

    // Map before skip.
    List<String> cdTitles1 = cdList
        .stream()                     // (1)
        .map(cd -> {                  // Map applied to all elements.
           System.out.println("Mapping: " + cd.title());
           return cd.title();
         })
        .skip(3)                      // Skip afterwards.
        .toList();
    System.out.println(cdTitles1);
    System.out.println();

    // Skip before map preferable.
    List<String> cdTitles2 = cdList
        .stream()                     // (2)
        .skip(3)                      // Skip first.
        .map(cd -> {                  // Map not applied to the first 3 element
           System.out.println("Mapping: " + cd.title());
           return cd.title();
         })
        .toList();
    System.out.println(cdTitles2);
  }
}
```

Output from the program:

Click here to view code image

```
Mapping: Java Jive
Mapping: Java Jam
Mapping: Lambda Dancing
Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]

Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]
```

Non-interfering and Stateless Behavioral Parameters

One of the main goals of the Stream API is that the code for a stream pipeline should execute and produce the same results whether the stream elements are processed sequentially or in parallel. In order to achieve this goal, certain constraints are placed on the *behavioral parameters*—that is, on the lambda expressions and method references that are implementations of the functional interface parameters in stream operations. These behavioral parameters, as the name implies, allow the behavior of a stream operation to be customized. For example, the predicate supplied to the `filter()` operation defines the criteria for filtering the elements.

Most stream operations require that their behavioral parameters are *non-interfering* and *stateless*. A *non-interfering behavioral parameter* does not change the stream data source during the execution of the pipeline, as this might not produce deterministic results. The exception to this is when the data source is *concurrent*, which guarantees that the source is thread-safe. A *stateless behavioral parameter* does not access any state that can change during the execution of the pipeline, as this might not be thread-safe.

If the constraints are violated, all bets are off, resulting in incorrect results being computed, which causes the stream pipeline to fail. In addition to these constraints, care should be taken to introduce side effects via behavioral parameters, as these might introduce other concurrency-related problems during parallel execution of the pipeline.

The aspects of intermediate operations mentioned in this subsection will become clearer as we fill in the details in subsequent sections.

Filtering

Filters are stream operations that select elements based on some criteria, usually specified as a *predicate*. This section discusses different ways of filtering elements, selecting unique elements, skipping elements at the head of a stream, and truncating a stream.

The following methods are defined in the `Stream<T>` interface, and analogous methods are also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> // Filtering using a predicate.
> Stream<T> filter(Predicate<? super T> predicate)
> ```
>
> Returns a stream consisting of the elements of this stream that match the given non-interfering, stateless predicate.
>
> This is a stateless intermediate operation that changes the stream size, but not the stream element type or the encounter order of the stream.

Click here to view code image

```
// Taking and dropping elements using a predicate.
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

The `takeWhile()` method puts an element from the input stream into its output stream, if it matches the predicate—that is, if the predicate returns the value `true` for this element. In this case, we say that the `takeWhile()` method *takes* the element.

The `dropWhile()` method discards an element from its input stream, if it matches the predicate—that is, if the predicate returns the value `true` for this element. In this case, we say that the `dropWhile()` method *drops* the element.

For an ordered stream:

The `takeWhile()` method takes elements from the input stream as long as an element matches the predicate, after which it short-circuits the stream processing.

The `dropWhile()` method drops elements from the input stream as long as an element matches the predicate, after which it passes through the remaining elements to the output stream.

In short, both methods find the *longest prefix of elements* to take or drop from the input stream, respectively.

For an unordered stream, where the predicate matches some but not all elements in the input stream:

The elements taken by the `takeWhile()` method or dropped by the `dropWhile()` method are *nondeterministic*; that is, any subset of matching elements can be taken or dropped, respectively, including the empty set.

If the predicate matches all elements in the input stream, regardless of whether the stream is ordered or unordered:

The `takeWhile()` method takes all elements; that is, the result is the same as the input stream.

The `dropWhile()` method drops all elements; that is, the result is the empty stream.

If the predicate matches no elements in the input stream, regardless of whether the stream is ordered or unordered:

The `takeWhile()` method takes no elements; that is, the result is the empty stream.

The `dropWhile()` method drops no elements; that is, the result is the same as the input stream.

Note that the `takeWhile()` method is a *short-circuiting* stateful intermediate operation, whereas the `dropWhile()` method is a stateful intermediate operation.

Click here to view code image

```
// Selecting distinct elements.
Stream<T> distinct()
```

Returns a stream consisting of the distinct elements of this stream, where no two elements are equal according to the `Object.equals()` method; that is, the method assumes that the elements override the `Object.equals()` method. It also uses the `hashCode()` method to keep track of the elements, and this method should also be overridden from the `Object` class.

For ordered streams, the *first* occurrence of a duplicated element is selected in the encounter order—called the *stability guarantee*. This stateful operation is particularly expensive for a parallel ordered stream which entails buffering overhead to ensure the stability guarantee. There is no such guarantee for an unordered stream: Which occurrence of a duplicated element will be selected is not guaranteed.

This stateful intermediate operation changes the stream size, but not the stream element type.

```
// Skipping elements.
Stream<T> skip(long n)
```

Returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream in encounter order. If this stream has fewer than `n` elements, an empty stream is returned.

This stateful operation is expensive for a parallel ordered stream which entails keeping track of skipping the *first* `n` elements.

This is a stateful intermediate operation that changes the stream size, but not the stream element type.

```
// Truncating a stream.
Stream<T> limit(long maxSize)
```

> Returns a stream consisting of elements from this stream, truncating the length of the returned stream to be no longer than the value of the parameter `maxSize`.
>
> This stateful operation is expensive for a parallel ordered stream which entails keeping track of passing the *first* `n` elements from the input stream to the output stream.
>
> This is a short-circuiting, stateful intermediate operation.

Filtering Using a Predicate

We have already seen many examples of filtering stream elements in this chapter. The first example of using the `Stream.filter()` method was presented in Figure 16.1, p. 885.

Filtering a collection using the `Iterator.remove()` method and the `Collection.removeIf()` method is discussed in §13.3, p. 691, and §15.2, p. 796, respectively.

The `filter()` method can be used on both object and numeric streams. The `Stream.filter()` method accepts a `Predicate<T>` as an argument. The predicate is typically implemented as a lambda expression or a method reference defining the selection criteria. It yields a stream consisting of elements from the input stream that satisfy the predicate. The elements that do not match the predicate are discarded.

In Figure 16.3, Query 1 selects those CDs from a list of CDs (`CD.cdList`) whose titles are in a set of popular CD titles (`popularTitles`). The `Collection.contains()` method is used in the predicate to determine if the title of a CD is in the set of popular CD titles. The execution of the stream pipeline shows there are only two such CDs (`cd0`, `cd1`). CDs that do not satisfy the predicate are discarded.

We can express the same query using the `Collection.removeIf()` method, as shown below. The code computes the same result as the stream pipeline in Figure 16.3. Note that the predicate in the `remove()` method call is a negation of the predicate in the `filter()` operation.

Click here to view code image

```
List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
System.out.println("Query 1b: " + popularCDs2);
//Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JA
```

In summary, the `filter()` method implements a stateless intermediate operation. It can change the size of the stream, since elements are discarded. However, the element type of the output stream returned by the `filter()` method is the same as that of its input stream. In Figure 16.3, the input and output stream type of the `filter()` method is `Stream<CD>`. Also, the encounter order of the stream remains unchanged. In Figure 16.3, the encounter order in the output stream returned by the `filter()` method is the same as the order of the elements in the input stream—that is, the insertion order in the list of CDs.

Figure 16.3 *Filtering Stream Elements*

Taking and Dropping Elements Using Predicates

Both the `takeWhile()` and the `dropWhile()` methods find the longest prefix of elements to take or drop from the input stream, respectively.

The code below at (1) and (2) illustrates the case for ordered streams. The `take-While()` method takes odd numbers from the input stream until a number is not odd, and short-circuits the processing of the stream—that is, it truncates the rest of the stream based on the predicate. The `dropWhile()` method, on the other hand, drops odd numbers from the input stream until a number is not odd, and passes the remaining elements to its output stream; that is, it skips elements in the beginning of the stream based on the predicate.

Click here to view code image

```
// Ordered stream:
Stream.of(1, 3, 5, 7, 8, 9, 11)                 // (1)

      .takeWhile(n -> n % 2 != 0)               // Takes longest prefix: 1 3 5
      .forEach(n -> System.out.print(n + " ")); // 1 3 5 7

Stream.of(1, 3, 5, 7, 8, 9, 11)                 // (2)
      .dropWhile(n -> n % 2 != 0)               // Drops longest prefix:  1 3
      .forEach(n -> System.out.print(n + " ")); // 8 9 11
```

Given an unordered stream, as shown below at (3), both methods return nondeterministic results: Any subset of matching elements can be taken or dropped, respectively.

Click here to view code image

```
// Unordered stream:
Set<Integer> iSeq = Set.of(1, 9, 4, 3, 7);      // (3)
iSeq.stream()
```

```
    .takeWhile(n -> n % 2 != 0)                  // Takes any subset of element
    .forEach(n -> System.out.print(n + " "));   // Nondeterministic: 1 9 7

iSeq.stream()
    .dropWhile(n -> n % 2 != 0)                  // Drops any subset of element
    .forEach(n -> System.out.print(n + " "));   // Nondeterministic: 4 3
```

Regardless of whether the stream is ordered or unordered, if *all* elements match the predicate, the `takeWhile()` method takes all the elements and the `dropWhile()` method drops all the elements, as shown below at (4) and (5).

Click here to view code image

```
// All match in ordered stream:                    (4)
Stream.of(1, 3, 5, 7, 9, 11)
      .takeWhile(n -> n % 2 != 0)                // Takes all elements.
      .forEach(n -> System.out.print(n + " ")); // Ordered: 1 3 5 7 9 11

Stream.of(1, 3, 5, 7, 9, 11)
      .dropWhile(n -> n % 2 != 0)                // Drops all elements.
      .forEach(n -> System.out.print(n + " ")); // Empty stream

// All match in unordered stream:                  (5)
Set<Integer> iSeq2 = Set.of(1, 9, 3, 7, 11, 5);
iSeq2.stream()
     .takeWhile(n -> n % 2 != 0)                 // Takes all elements.
     .forEach(n -> System.out.print(n + " "));  // Unordered: 9 11 1 3 5 7

iSeq2.stream()
     .dropWhile(n -> n % 2 != 0)                 // Drops all elements.
     .forEach(n -> System.out.print(n + " "));  // Empty stream
```

Regardless of whether the stream is ordered or unordered, if *no* elements match the predicate, the `takeWhile()` method takes no elements and the `dropWhile()` method drops no elements, as shown below at (6) and (7).

Click here to view code image

```
// No match in ordered stream:                     (6)
Stream.of(2, 4, 6, 8, 10, 12)
      .takeWhile(n -> n % 2 != 0)                // Takes no elements.
      .forEach(n -> System.out.print(n + " ")); // Empty stream

Stream.of(2, 4, 6, 8, 10, 12)
      .dropWhile(n -> n % 2 != 0)                // Drops no elements.
      .forEach(n -> System.out.print(n + " ")); // Ordered: 2 4 6 8 10 12


// No match in unordered stream:                   (7)
Set<Integer> iSeq3 = Set.of(2, 10, 8, 12, 4, 6);
iSeq3.stream()
     .takeWhile(n -> n % 2 != 0)                 // Takes no elements.
     .forEach(n -> System.out.print(n + " "));  // Empty stream

iSeq3.stream()
     .dropWhile(n -> n % 2 != 0)                 // Drops no elements.
```

```
        .forEach(n -> System.out.print(n + " "));  // Unordered: 8 10 12 2 4 6
```

Selecting Distinct Elements

The `distinct()` method removes all duplicates of an element from the input stream, resulting in an output stream with only unique elements. Since the `distinct()` method must be able to distinguish the elements from one another and keep track of them, the stream elements must override the `equals()` and the `hashCode()` methods of the `Object` class. The `CD` objects comply with this requirement (Example 16.1, p. 883).

In Figure 16.4, Query 2 creates a list of unique CDs with pop music. The `filter()` operation and the `distinct()` operation in the stream pipeline select the CDs with pop music and those that are unique, respectively. The execution of the stream pipeline shows that the resulting list of unique CDs with pop music has only one CD (`cd0`).

In Figure 16.4, interchanging the stateless `filter()` operation and the stateful `distinct()` operation in the stream pipeline gives the same results, but then the more expensive `distinct()` operation is performed on *all* elements of the stream, rather than on a shorter stream which is returned by the `filter()` operation.

Figure 16.4 *Selecting Distinct Elements*

Skipping Elements in a Stream

The `skip()` operation slices off or discards a specified number of elements from the head of a stream before the remaining elements are made available to the next operation. It preserves the encounter order if the input stream has one. Not surprisingly, skipping more elements than are in the input stream returns the empty stream.

In Figure 16.5, Query 3a creates a list of jazz music CDs after skipping the first two CDs in the stream. The stream pipeline uses a `skip()` operation first to discard two CDs (one of them being a jazz music CD) and a `filter()` operation afterward to select any CDs with jazz music. The execution of this stream pipeline shows that the resulting list contains two CDs (`cd3`, `cd4`).

In the stream pipeline in Figure 16.5, the `skip()` operation is before the `filter()` operation. Switching the order of the `skip()` and `filter()` operations as in Query 3b in Example 16.5 does not solve the same query. It will skip the first two jazz music CDs selected by the `filter()` operation.

Figure 16.5 *Skipping Elements at the Head of a Stream*

Truncating a Stream

The `limit()` operation returns an output stream whose maximum size is equal to the max size specified as an argument to the method. The input stream is only truncated if its size is greater than the specified max size.

In Figure 16.6, Query 4 creates a list with the first two CDs that were released in 2018. The stream pipeline uses a `filter()` operation first to select CDs released in 2018, and the `limit()` operation truncates the stream, if necessary, so that, at most, only two CDs are passed to its output stream. The short-circuit execution of this stream pipeline is illustrated in Figure 16.6, showing the resulting list containing two CDs (`cd2`, `cd3`). The execution of the stream pipeline terminates after the `limit()` operation has reached its limit if there are no more elements left to process. In Figure 16.6, we can see that the limit was reached and execution was terminated. Regardless of the fact that the last element in the initial stream was not processed, the stream cannot be reused once the execution of the pipeline terminates due to a short-circuiting operation.

Figure 16.6 *Truncating a Stream*

The `limit()` operation is ideal for turning an infinite stream into a finite stream. Numerous examples of using the `limit()` operation with the `iterate()` and `generate()` methods can be found in §16.4, p. 894, and with the `Random.ints()` method in §16.4, p. 900.

For a given value `n`, `limit(n)` and `skip(n)` are complementary operations on a stream, as `limit(n)` comprises the first `n` elements of the stream and `skip(n)` comprises the remaining elements in the stream. In the code below, the `resultList` from processing the resulting stream from concatenating the two substreams is equal to the stream source `CD.cdList`.

Click here to view code image

```
List<CD> resultList = Stream
    .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
    .toList();
System.out.println(CD.cdList.equals(resultList));            // true
```

The `skip()` operation can be used in conjunction with the `limit()`

operation to process a substream of a stream, where the `skip()` operation can be used to skip to the start of the substream and the `limit()` operation to limit the size of the substream. The substream in the code below starts at the second element and comprises the next three elements in the stream.

Click here to view code image

```
List<CD> substream = CD.cdList
    .stream()
    .skip(1)
    .limit(3)
    .toList();
System.out.println("Query 5: " + substream);
// Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
//           <Funkies, "Lambda Dancing", 10, 2018, POP>,
//           <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

The `limit()` operation is a short-circuiting stateful intermediate operation, as it needs to keep state for tracking the number of elements in the output stream. It changes the stream size, but not the stream element type or the encounter order. For an ordered stream, we can expect the elements in the resulting stream to have the same order, but we cannot assume any order if the input stream is unordered.

Example 16.5 contains the code snippets presented in this subsection.

Example 16.5 *Filtering*

Click here to view code image

```
import java.time.Year;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Stream;

public final class Filtering {
  public static void main(String[] args) {

    // Query 1: Find CDs whose titles are in the set of popular CD titles.
    Set<String> popularTitles = Set.of("Java Jive", "Java Jazz", "Java Jam");

    // Using Stream.filter().
    List<CD> popularCDs1 = CD.cdList
        .stream()
        .filter(cd -> popularTitles.contains(cd.title()))
        .toList();
    System.out.println("Query 1a: " + popularCDs1);

    // Using Collection.removeIf().
    List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
    popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
    System.out.println("Query 1b: " + popularCDs2);

    // Query 2: Create a list of unique CDs with pop music.
    List<CD> miscCDList = List.of(CD.cd0, CD.cd0, CD.cd1, CD.cd0);
    List<CD> uniquePopCDs1 = miscCDList
        .stream()
        .filter(CD::isPop)
        .distinct()                                // distinct() after filter(
        .toList();
    System.out.println("Query 2: " + uniquePopCDs1);

    // Query 3a: Create a list of jazz CDs, after skipping the first two CDs.
    List<CD> jazzCDs1 = CD.cdList
        .stream()
        .skip(2)                                   // skip() before filter().
        .filter(CD::isJazz)
        .toList();
    System.out.println("Query 3a: " + jazzCDs1);

    // Query 3b: Create a list of jazz CDs, but skip the first two jazz CDs.
    List<CD> jazzCDs2 = CD.cdList                  // Not equivalent to Query
        .stream()
        .filter(CD::isJazz)
        .skip(2)                                   // skip() after filter().
        .toList();
    System.out.println("Query 3b: " + jazzCDs2);

    // Query 4: Create a list with the first 2 CDs that were released in 2018.
    List<CD> twoFirstCDs2018 = CD.cdList
        .stream()
        .filter(cd -> cd.year().equals(Year.of(2018)))
        .limit(2)
        .toList();
    System.out.println("Query 4: " + twoFirstCDs2018);

    // limit(n) and skip(n) are complementary.
    List<CD> resultList = Stream
        .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
        .toList();
    System.out.println(CD.cdList.equals(resultList));

    // Query 5: Process a substream by skipping 1 and limiting the size to 3.
    List<CD> substream = CD.cdList
        .stream()
        .skip(1)
        .limit(3)
        .toList();
    System.out.println("Query 5: " + substream);
  }
}
```

Output from the program (*formatted to fit on the page*):

Click here to view code image

```
Query 1a: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ
Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ
Query 2: [<Jaav, "Java Jive", 8, 2017, POP>]
Query 3a: [<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
           <Genericos, "Hot Generics", 10, 2018, JAZZ>]
Query 3b: [<Genericos, "Hot Generics", 10, 2018, JAZZ>]
Query 4: [<Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

```
true
Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
          <Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

Examining Elements in a Stream

The `peek()` operation allows stream elements to be examined at the point where the operation is used in the stream pipeline. It does not affect the stream in any way, as it only facilitates a side effect via a non-interfering *consumer* specified as an argument to the operation. It is primarily used for debugging the pipeline by examining the elements at various points in the pipeline.

The following method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> Stream<T> peek(Consumer<? super T> action)
> ```
>
> Returns a stream consisting of the same elements as those in this stream, but additionally performs the provided non-interfering `action` on each element as elements are processed from this stream.
>
> This is a stateless intermediate operation that does not change the stream size, the stream element type, or the encounter order.

By using the `peek()` method, we can dispense with explicit print statements that were inserted in the implementation of the behavioral parameter of the `map()` operation in Example 16.4, p. 909. Example 16.6 shows how the `peek()` operation can be used to trace the processing of elements in the pipeline. A `peek()` operation after each intermediate operation prints pertinent information which can be used to verify the workings of the pipeline. In Example 16.6, the output shows that the `skip()` operation before the `map()` operation can improve performance, as the `skip()` operation shortens the stream on which the `map()` operation should be performed.

Example 16.6 *Examining Stream Elements*

Click here to view code image

```
import java.util.List;

public final class OrderOfOperationsWithPeek {
  public static void main(String[] args) {

    System.out.println("map() before skip():");
    List<String> cdTitles1 = CD.cdList
        .stream()
        .map(CD::title)
        .peek(t -> System.out.println("After map: " + t))
        .skip(3)
        .peek(t -> System.out.println("After skip: " + t))
        .toList();
    System.out.println(cdTitles1);
    System.out.println();

    System.out.println("skip() before map():");            // Preferable.
    List<String> cdTitles2 = CD.cdList
        .stream()
        .skip(3)
        .peek(cd -> System.out.println("After skip: " + cd))
        .map(CD::title)
        .peek(t -> System.out.println("After map: " + t))
        .toList();
    System.out.println(cdTitles2);
  }
}
```

Output from the program:

Click here to view code image

```
map() before skip():
After map: Java Jive
After map: Java Jam
After map: Lambda Dancing
After map: Keep on Erasing
After skip: Keep on Erasing
After map: Hot Generics
After skip: Hot Generics
[Keep on Erasing, Hot Generics]

skip() before map():
After skip: <Genericos, "Keep on Erasing", 8, 2018, JAZZ>
After map: Keep on Erasing
After skip: <Genericos, "Hot Generics", 10, 2018, JAZZ>
After map: Hot Generics
[Keep on Erasing, Hot Generics]
```

Mapping: Transforming Streams

The `map()` operation has already been used in several examples (Example 16.3, p. 906, Example 16.4, p. 909, and Example 16.6, p. 920). Here we take a closer look at this essential intermediate operation for data processing using a stream. It maps one type of stream (`Stream<T>`) into another type of stream (`Stream<R>`); that is, *each* element of type `T` in the input stream is mapped to an element of type `R` in the output stream by the function (`Function<T, R>`) supplied to the `map()` method. It defines a *one-to-one mapping*. For example, if we are interested in the *titles* of CDs in the CD stream, we can use the `map()` operation to transform each `CD` in the stream to a `String` that represents the title of the `CD` by applying an appropriate function:

Click here to view code image

```
Stream<String> titles = CD.cdList
         .stream()                          // Input stream: Stream<CD>.
         .map(CD::title);                   // Lambda expression: cd -> cd.title()
```

The following methods are defined in the `Stream<T>` interface, and analogous methods are also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> <R> Stream<R> map(Function<? super T,? extends R> mapper)
> ```
>
> Returns a stream consisting of the result of applying the given non-interfering, stateless function to the elements of this stream—that is, it creates a new stream (`Stream<R>`) from the results of applying the `mapper` function to the elements of this stream (`Stream<T>`).
>
> This is an intermediate operation that does not change the stream size, but it can change the stream element type and does not guarantee to preserve the encounter order of the input stream.
>
> Click here to view code image
>
> ```
> // Converting Stream<T> to a Numeric Stream
> IntStream    mapToInt(ToIntFunction<? super T> mapper)
> LongStream   mapToLong(ToLongFunction<? super T> mapper)
> DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
> ```
>
> Return the numeric stream consisting of the results of applying the given non-interfering, stateless function to the elements of this stream—that is, they create a stream of numeric values that are the results of applying the `mapper` function to the elements of this stream (`Stream<T>`).
>
> These operations are all intermediate operations that transform an object stream to a numeric stream. The stream size is not affected, but there is no guarantee that the encounter order of the input stream is preserved.

In Figure 16.7, the query creates a list with CD titles released in 2018. The stream pipeline uses a `filter()` operation first to select

CDs released in 2018, and the `map()` operation maps a `CD` to its title (`String`). The input stream is transformed by the `map()` operation from `Stream<CD>` to `Stream<String>`. The execution of this stream pipeline shows the resulting list (`List<String>`) containing three CD titles.

Figure 16.7 *Mapping*

The query below illustrates transforming an object stream to a numeric stream. When executed, the stream pipeline prints the years in which the CDs were released. Note the transformation of the initial stream, `Stream<CD>`. The `map()` operation first transforms it to a `Stream<Year>` and the `distinct()` operation selects the unique years. The `mapToInt()` operation transforms the stream from `Stream<Year>` to `IntStream`—that is, a stream of `int`s whose values are then printed.

Click here to view code image

```
CD.cdList.stream()                                    // Stream<CD>
         .map(CD::year)                               // Stream<Year>
         .distinct()                                  // Stream<Year>
         .mapToInt(Year::getValue)                    // IntStream
         .forEach(System.out::println);               // 2017
                                                      // 2018
```

In the example below, the `range()` method generates an `int` stream for values in the half-open interval specified by its arguments. The values are generated in *increasing* order, starting with the lower bound of the interval. In order to generate them in *decreasing* order, the `map()` operation can be used to reverse the values. In this case, the input stream and output stream of the `map()` operation are both `IntStream`s.

Click here to view code image

```
int from = 0, to = 5;
IntStream.range(from, to)                    // [0, 5)
         .map(i -> to + from – 1 – i)        // Reverse the stream values

         .forEach(System.out::print);        // 43210
```

The stream pipeline below determines the number of times the dice value is 6. The `generate()` method generates a value between 1 and 6, and the `limit()` operation limits the max size of the stream. The `map()` operation returns the value 1 if the dice value is 6; otherwise, it returns 0. In other words, the value of the dice throw is mapped either to 1 or 0, depending on the dice value. The terminal operation `sum()` sums the values in the streams, which in this case are either 1 or 0, thus returning the correct number of times the dice value was 6.

Click here to view code image

```
long sixes = IntStream
   .generate(() -> (int) (6.0 * Math.random()) + 1) // [1, 6]
   .limit(2000)                                     // Number of throws.
   .map(i -> i == 6 ? 1 : 0)             // Dice value mapped to 1 or 0.
   .sum();
```

Flattening Streams

The `flatMap()` operation first maps each element in the input stream to *a mapped stream*, and then *flattens* the mapped streams to a *single* stream—that is, the elements of each mapped stream are incorporated into a single stream when the pipeline is executed. In other words, each element in the input stream may be mapped to many elements in the output stream. The `flatMap()` operation thus defines a *one-to-many mapping* that flattens a multilevel stream by one level.

The following method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> <R> Stream<R> flatMap(
>                   Function<? super T,? extends Stream<? extends R>> mapper)
> ```
>
> The `mapper` function maps each element of type `T` in this stream to a *mapped stream* (`Stream<R>`). The method returns an output stream (`Stream<R>`) which is the result of replacing each element of type `T` in this stream with the *elements* of type `R` from its mapped stream.
>
> If the result of the `mapper` function is `null`, the empty stream is used as the mapped stream.
>
> This is an intermediate operation that changes the stream size and the element type of the stream, and does not guarantee to preserve the encounter order of the input stream.

The methods below are defined only in the `Stream<T>` interface. No counterparts exist in the `IntStream`, `LongStream`, or `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> IntStream    flatMapToInt(Function<? super T,? extends IntStream> mapper)
> ```

```
LongStream   flatMapToLong(Function<? super T,? extends LongStream> mapper)
DoubleStream flatMapToDouble(
                 Function<? super T,? extends DoubleStream> mapper)
```

> The `mapper` function maps each element of type `T` in this stream to a *mapped numeric stream* (*NumType*`Stream`). The method returns an output stream (*NumType*-`Stream`), which is the result of replacing each element of type `T` in this stream with the *values* from its mapped numeric stream. The designation *`NumType`* stands for `Int`, `Long`, or `Double`.

To motivate using the `flatMap()` operation, we look at how to express the query for creating a list of unique CDs from two given lists of CDs. Figure 16.8 shows an attempt to express this query by creating a stream of lists of CDs, `Stream<List<CD>>`, and selecting the unique CDs using the `distinct()` method. This attempt fails miserably as the `distinct()` method distinguishes between elements that are lists of CDs, and not individual CDs. Figure 16.8 shows the execution of the stream pipeline resulting in a list of lists of CDs, `List<List<CD>>`.

Figure 16.8 *Incorrect Solution to the Query*

The next attempt to express the query uses the `map()` operation as shown in Figure 16.9. The idea is to map each list of CDs (`List<CD>`) to a stream of CDs (`Stream<CD>`), and select the unique CDs with the `distinct()` operation. The mapper function of the `map()` operation maps each list of CDs to *a mapped stream* that is a stream of CDs, `Stream<CD>`. The resulting stream from the `map()` operation is a stream of streams of CDs, `Stream<Stream<CD>>`. The `distinct()` method distinguishes between elements that are mapped streams of CDs. Figure 16.9 shows the execution of the stream pipeline resulting in a list of mapped streams of CDs, `List<Stream<CD>>`.

Figure 16.9 *Mapping a Stream of Streams*

The `flatMap()` operation provides the solution, as it flattens the contents of the mapped streams into a single stream so that the `distinct()` operation can select the unique CDs individually. The stream pipeline using the `flatMap()` operation and its execution are shown in Figure 16.10. The mapper function of the `flatMap()` operation maps each list of CDs to a mapped stream that is a stream of CDs, `Stream<CD>`. The contents of the mapped stream are flattened into the output stream. The resulting stream from the `flatMap()` operation is a stream of CDs, `Stream<CD>`. Note how each list in the initial stream results in a flattened stream whose elements are processed by the pipeline. The result list of CDs contains the unique CDs from the two lists.

Figure 16.10 *Flattening Streams*

The code below flattens a two-dimensional array to a one-dimensional array. The `Arrays.stream()` method call at (1) creates an object stream, `Stream<int[]>`, whose elements are arrays that are rows in the two-dimensional array. The mapper of the `flatMapToInt()` operation maps each row in the `Stream<int[]>` to a stream of `int`s (`IntStream`) by applying the `Array.stream()` method at (2) to each row. This would result in a stream of mapped streams of `int`s (`Stream<IntStream>>`), but it is flattened by the `flatMapToInt()` operation to a final stream of `int`s (`IntStream`). The terminal operation `toArray()` creates an appropriate array in which the `int` values of the final stream are stored (p. 971).

Click here to view code image

```
int[][] twoDimArray = { {2017, 2018}, {1948, 1949} };
int[] intArray = Arrays
    .stream(twoDimArray)                         // (1) Stream<int[]>

    .flatMapToInt(row -> Arrays.stream(row)) // (2) mapper: int[] -> IntStream
                                       // flattens Stream<IntStream> to IntStre
    .toArray();                                  // [2017, 2018, 1948, 1949]
```

Replacing Each Element of a Stream with Multiple Elements

The `mapMulti()` intermediate operation applies a *one-to-many transformation* to the elements of the stream and flattens the result elements into a new stream. The functionality of the `mapMulti()` method is very similar to that of the `flatMap()` method. Whereas the latter uses a `Function<T, Stream<R>>` mapper to create a *mapping stream* for each element and then flattens the stream, the former applies a `BiConsumer<T, Consumer<R>>` mapper to each element. The mapper calls the `Consumer` to accept the replacement elements that are incorporated into a single stream when the pipeline is executed.

The `mapMulti()` method can be used to perform filtering, mapping, and flat mapping of stream elements, all depending on the implementation of the `BiConsumer` mapper passed to the method.

The code below shows a *one-to-one transformation* of the stream elements. A `BiConsumer` is defined at (1) that first filters the stream for pop music CDs at (2), and maps each CD to a string that contains its title and its number of tracks represented by an equivalent number of `"*"` characters. The resulting string is submitted at (3) to the consumer (supplied by the `mapMulti()` method). Each value passed to the `accept()` method of the consumer replaces the current element in the stream. Note that the body of the `BiConsumer` is implemented in an *imperative* manner using an `if` statement. The `BiConsumer` created at (1) is passed to the `mapMulti()` method at (5) to process the CDs of the stream created at (4). The `mapMulti()` method passes an appropriate `Consumer` to the `BiConsumer` that accepts the replacement elements.

Click here to view code image

```
// One-to-one
BiConsumer<CD, Consumer<String>> bcA = (cd, consumer) -> {              // (1)
  if (cd.genre() == Genre.POP) {                                        // (2)
    consumer.accept(String.format("%-15s: %s", cd.title(),              // (3)
                                  "*".repeat(cd.noOfTracks())));
  }
};

CD.cdList.stream()                                                      // (4)
          .mapMulti(bcA)                                                // (5)
          .forEach(System.out::println);
```

Output from the code:

```
Java Jive      : ********
Lambda Dancing : **********
```

The code below shows a *one-to-many* transformation of the stream elements. The `BiConsumer` at (1) iterates through a list of CDs and maps each CD in the list to its title. Each list of CDs in the stream will thus be replaced with the titles of the CDs in the list. The `mapMulti()` operation with the `BiConsumer` at (1) is applied at (3) to a stream of list of CDs (`Stream<List<CD>>`) created at (2). The `mapMulti()` operation in this case is analogous to the `flatMap()` operation to achieve the same result.

Click here to view code image

```
// One-to-many
List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1);
List<CD> cdList2 = List.of(CD.cd0, CD.cd1);
BiConsumer<List<CD>, Consumer<String>> bcB = (lst, consumer) -> {       // (1)
  for (CD cd : lst) {
    consumer.accept(cd.title());
  }
};
List<String> listOfCDTitles = Stream.of(cdList1, cdList2) // (2) Stream<List<C
    .mapMulti(bcB)                                        // (3)
    .distinct()
    .toList();
System.out.println(listOfCDTitles);                       // [Java Jive, Java
```

The previous two code snippets first defined the `BiConsumer` with all relevant types specified explicitly, and then passed it to the `mapMulti()` method. The code below defines the implementation of the `BiConsumer` in the call to the `mapMulti()` method. We consider three alternative implementations as exemplified by (2a), (2b), and (2c).

Alternative (2a) results in a compile-time error. The reason is that the compiler cannot unequivocally infer the actual type parameter `R` of the consumer parameter of the lambda expression. It can only infer that the type of the `lst` parameter is `List<CD>` as it denotes an element of stream whose type is `Stream<List<CD>>`. The compiler makes the safest assumption that the type parameter `R` is `Object`. With this assumption, the resulting list is of type `List<Object>`, but this cannot be assigned to a reference of type `List<String>`, as declared in the assignment statement. To avoid the compile-time error in this case, we can change the type of the reference to `Object` or to the wildcard `?`.

Alternative (2b) uses the type witness `<String>` in the call to the `mapMulti()` method to explicitly corroborate the actual type parameter of the consumer.

Alternative (2c) explicitly specifies the types for the parameters of the lambda expression.

Click here to view code image

```
List<String> listOfCDTitles2 = Stream.of(cdList1,cdList2) // (1) Stream<List<C
//  .mapMulti((lst, consumer) -> {                    // (2a) Compile-time err
//  .<String>mapMulti((lst, consumer) -> {                     // (2b) OK.
    .mapMulti((List<CD> lst, Consumer<String> consumer) -> {   // (2c) OK.
      for (CD cd : lst) {
        consumer.accept(cd.title());
      }
    })
    .distinct()
    .toList();
System.out.println(listOfCDTitles2);                  // [Java Jive, Java Jam]
```

The `mapMulti()` method is preferable to the `flatMap()` method under the following circumstances:

- When an element is to be replaced with a small number of elements, or none at all. The `mapMulti()` method avoids the overhead of creating a mapped stream for each element, as done by the `flatMap()` method.
- When an *imperative approach* for creating replacement elements is easier than using a stream.

The following default method is defined in the `Stream<T>` interface, and an analogous method is also defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> default <R> Stream<R> mapMulti(
>                  BiConsumer<? super T,? super Consumer<R>> mapper)
> ```
>
> Returns a stream that is a result of replacing each element of this stream with multiple elements, specifically zero or more elements.
>
> The specified `mapper` is applied to each element in conjunction with a *consumer* that *accepts* replacement elements. The `mapper` calls the consumer zero or more times to accept the replacement elements.
>
> Note that the consumer is supplied by the `mapMulti()` method, and called by the `mapper` to accept replacement elements. An element of type `T` is replaced with zero or more elements of type `R`.
>
> This is an intermediate operation that changes the stream size and the element type of the stream, and does not guarantee to preserve the encounter order of the input stream.

The following default methods are defined only in the `Stream<T>` interface. No counterparts exist in the `IntStream`, `LongStream`, or `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> default IntStream
>        mapMultiToInt(BiConsumer<? super T,? super IntConsumer> mapper)
> default LongStream
>        mapMultiToLong(BiConsumer<? super T,? super LongConsumer> mapper)
> default DoubleStream
>        mapMultiToDouble(BiConsumer<? super T,? super DoubleConsumer> mapper)
> ```
>
> Return an `IntStream`, `LongStream`, and `DoubleStream`, respectively, consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements.

Sorted Streams

The `sorted()` intermediate operation can be used to enforce a specific encounter order on the elements of the stream. It is important to note that the data source is *not* sorted; only the order of the elements in the stream is affected when a stream is sorted. It is an expensive stateful operation, as state must be kept for all elements in the stream before making them available in the resulting stream.

The following methods are defined in the `Stream<T>` interface, but only the first method is defined in the `IntStream`, `LongStream`, and `DoubleStream` interfaces:

> Click here to view code image
>
> ```
> Stream<T> sorted()
> Stream<T> sorted(Comparator<? super T> cmp)
> ```
>
> Return a stream containing the elements of this stream, sorted according to natural order or according to total order defined by the specified comparator, respectively.
>
> The first method requires that the elements implement the `Comparable<T>` interface.
>
> The sorting operation provides a *stability guarantee* for ordered streams only— that is, duplicates of an element will be in their encounter order in the resulting stream.
>
> This is a stateful intermediate operation that does not change the size of the stream or the stream element type, and enforces the sort order to be the encounter order of the resulting stream.

The `Comparable<E>` and `Comparator<E>` interfaces are covered in §14.4, p. 761, and §14.5, p. 769, respectively.

Example 16.7 illustrates the `sorted()` operation on streams. Printing the array at (1) and executing the stream pipeline at (2) shows that the order of the elements in the array and in the stream is *positional order*, as one would expect. The zero-argument `sorted()` method sorts in *natural order*, as in the pipeline at (3). It expects the stream elements to implement the `Comparable<CD>` interface. The `sorted()` method in the pipeline at (4) uses the *reverse natural order* to sort the elements.

The pipeline at (5) represents the query to find all jazz music CDs and sort them by their title. A comparator to compare by title is passed to the `sorted()` method. Finally, the pipeline at (6) finds CDs with eight or more tracks, and sorts them according to the number of tracks. An appropriate comparator that compares by the number of tracks is passed to the `sorted()` method.

It is instructive to compare the output showing the results from each pipeline in Example 16.7. The comparators in Example 16.7 are also implemented as lambda expressions, in addition to their implementation by the methods in the `Comparator<E>` interface.

Example 16.7 *Sorting Streams*

Click here to view code image

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Sorting {
  public static void main(String[] args) {

    System.out.println("(1) Positional order in the array:");

    CD[] cdArray = CD.cdArray;
    System.out.println(Arrays.toString(cdArray));               // (1)

    System.out.println("(2) Positional order in the stream:");
    List<CD> cdsByPositionalOrder =                             // (2)
      Arrays.stream(cdArray)
            .toList();
    System.out.println(cdsByPositionalOrder);

    System.out.println("(3) Natural order:");
    List<CD> cdsByNaturalOrder =                                // (3)
      Arrays.stream(cdArray)
            .sorted()
            .toList();
    System.out.println(cdsByNaturalOrder);

    System.out.println("(4) Reversed natural order:");
    List<CD> cdsByRNO =                                         // (4)
      Arrays.stream(cdArray)
//          .sorted((c1, c2) -> -c1.compareTo(c2))
            .sorted(Comparator.reverseOrder())
            .toList();
    System.out.println(cdsByRNO);

    System.out.println("(5) Only Jazz CDs, ordered by title:");
    List<String> jazzCDsByTitle =                               // (5)
      Arrays.stream(cdArray)
            .filter(CD::isJazz)
//          .sorted((c1, c2) -> c1.title().compareTo(c2.title()))
            .sorted(Comparator.comparing(CD::title))
            .map(CD::title)
            .toList();
    System.out.println(jazzCDsByTitle);

    System.out.println("(6) No. of tracks >= 8, ordered by number of tracks:")
    List<CD> cds =                                              // (6)
      Arrays.stream(cdArray)
            .filter(d -> d.noOfTracks() >= 8)
//          .sorted((c1, c2) -> c1.noOfTracks() - c2.noOfTracks())
```

```
                .sorted(Comparator.comparing(CD::noOfTracks))
                .toList();
        System.out.println(cds);
    }
  }
```

Output from the program (*formatted to fit on the page*):

Click here to view code image

```
(1) Positional order in the array:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
(2) Positional order in the stream:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,

 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
(3) Natural order:
[<Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Jaav, "Java Jive", 8, 2017, POP>]
(4) Reversed natural order:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>]
(5) Only Jazz CDs, ordered by title:
[Hot Generics, Java Jam, Keep on Erasing]
(6) No. of tracks >= 8, ordered by number of tracks:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
```

Setting a Stream as Unordered

The `unordered()` intermediate operation does not actually reorder the elements in the stream to make them unordered. It just removes *the ordered constraint* on a stream if this constraint is set for the stream, indicating that stream operations can choose to ignore its encounter order. Indicating the stream to be unordered can improve the performance of some operations. For example, the `limit()`, `skip()`, and `distinct()` operations can improve performance when executed on unordered parallel streams, since they can process *any* elements by ignoring the encounter order. The removal of the ordered constraint can impact the performance of certain operations on parallel streams (p. 1015).

It clearly makes sense to call the `unordered()` method on an ordered stream only if the order is of no consequence in the final result. There is no method called `ordered` to impose an order on a stream. However, the `sorted()` intermediate operation can be used to enforce a sort order on the output stream.

In the stream pipeline below, the `unordered()` method clears the ordered constraint on the stream whose elements have the same order as in the data source—that is, the positional order in the list of CDs. The outcome of the execution shows that the titles in the result list are in the same order as they are in the data source; this is the same result one would get without the `unordered()` operation. It is up to the stream operation to take into consideration that the stream is unordered. The fact that the result list retains the order does not make it invalid. After all, since the stream is set as unordered, it indicates that ignoring the order is at the discretion of the stream operation.

Click here to view code image

```
//Query: Create a list with the first 2 Jazz CD titles.
List<String> first2JazzCDTitles = CD.cdList
    .stream()
    .unordered()                     // Don't care about ordering.
    .filter(CD::isJazz)
    .limit(2)
    .map(CD::title)
    .toList();                       // [Java Jam, Keep on Erasing]
```

The following method is inherited by the `Stream<T>` interface from its superinterface `BaseStream`. Analogous methods are also inherited by the `IntStream`, `LongStream`, and `DoubleStream` interfaces from the superinterface `BaseStream`.

> `Stream<T> unordered()`
>
> Returns an unordered sequential stream that has the same elements as this stream. The method returns itself if this stream is already unordered. The method can only indicate that the encounter order of this stream can be ignored, and an operation might not comply to this request.
>
> This is an intermediate operation that does not change the stream size or the stream element type. It only indicates that the encounter order can be ignored.

Execution Mode of a Stream

The two methods `parallel()` and `sequential()` are intermediate operations that can be used to set the execution mode of a stream—

that is, whether it will execute sequentially or in parallel. Only the `Collection.parallelStream()` method creates a parallel stream from a collection, so the default mode of execution for most streams is sequential, unless the mode is specifically changed by calling the `parallel()` method. The execution mode of a stream can be switched between sequential and parallel execution at any point between stream creation and the terminal operation in the pipeline. However, it is the *last* call to any of these methods that determines the execution mode for the *entire* pipeline, regardless of how many times these methods are called in the pipeline.

The declaration statements below show examples of both sequential and parallel streams. No stream pipeline is executed, as no terminal operation is invoked on any of the streams. However, when a terminal operation is invoked on one of the streams, the stream will be executed in the mode indicated for the stream.

Click here to view code image

```
Stream<CD> seqStream1
  = CD.cdList.stream().filter(CD::isPop);                           // Sequentia
Stream<CD> seqStream2
  = CD.cdList.stream().sequential().filter(CD::isPop);              // Sequentia
Stream<CD> seqStream3
  = CD.cdList.stream().parallel().filter(CD::isPop).sequential(); // Sequentia
Stream<CD> paraStream1
  = CD.cdList.stream().parallel().filter(CD::isPop);                // Parallel
Stream<CD> paraStream2
  = CD.cdList.stream().filter(CD::isPop).parallel();                // Parallel
```

The `isParallel()` method can be used to determine the execution mode of a stream. For example, the call to the `isParallel()` method on `seqStream3` below shows that this stream is a sequential stream. It is the call to the `sequential()` method that occurs last in the pipeline that determines the execution mode.

Click here to view code image

```
System.out.println(seqStream3.isParallel());                        // false
```

Parallel streams are explored further in §16.9, p. 1009.

The following methods are inherited by the `Stream<T>` interface from its superinterface `BaseStream`. Analogous methods are also inherited by the `IntStream`, `LongStream`, and `DoubleStream` interfaces from the superinterface `BaseStream`.

```
Stream<T> parallel()
Stream<T> sequential()
```

Set the execution mode of a stream. They return a parallel or a sequential stream that has the same elements as this stream, respectively. Each method will return itself if this stream is already parallel or sequential, respectively.

These are intermediate operations that do not change the stream size, the stream element type, or the encounter order.

```
boolean isParallel()
```

Returns whether this stream would execute in parallel when the terminal operation is invoked. The method might yield unpredictable results if called after a terminal operation has been invoked.

It is *not* an intermediate operation.

Converting between Stream Types

Table 16.2 provides a summary of interoperability between stream types—that is, transforming between different stream types. Where necessary, the methods are shown with the name of the built-in functional interface required as a parameter. Selecting a naming convention for method names makes it easy to select the right method for transforming one stream type to another.

Table 16.2 *Interoperability between Stream Types*

Stream type [illegible]				
From Stream<T> [illegible] `ToLongFunction)` [illegible]				
From IntStream [illegible] [illegible]				
From LongStream [illegible] [illegible]				
From DoubleStream [illegible] [illegible]				

Mapping between Object Streams

The `map()` and `flatMap()` methods of the `Stream<T>` interface transform an object stream of type `T` to an object stream of type `R`. Examples using these two methods can be found in §16.5, p. 921, and §16.5, p. 924, respectively.

Mapping an Object Stream to a Numeric Stream

The `mapTo`*NumType*`()` methods in the `Stream<T>` interface transform an object stream to a stream of the designated numeric type, where *`NumType`* is either `Int`, `Long`, or `Double`.

The query below sums the number of tracks for all CDs in a list. The `mapToInt()` intermediate operation at (2) accepts an `IntFunction` that extracts the number of tracks in a `CD`, thereby transforming the `Stream<CD>` created at (1) into an `IntStream`. The terminal operation `sum()`, as the name implies, sums the values in the `IntStream` (p. 973).

Click here to view code image

```
int totalNumOfTracks = CD.cdList
    .stream()                                         // (1) Stream<CD>
    .mapToInt(CD::noOfTracks)                         // (2) IntStream
    .sum();                                           // 42
```

The `flatMapTo`*NumType*`()` methods are only defined by the `Stream<T>` interface to flatten a multilevel object stream to a numeric stream, where *`NumType`* is either `Int`, `Long`, or `Double`.

Earlier we saw an example of flattening a two-dimensional array using the `flat-MapToInt()` method (p. 924).

The query below sums the number of tracks for all CDs in *two* `CD` lists. The `flatMapToInt()` intermediate operation at (1) accepts a `Function` that maps each `List<CD>` in a `Stream<List<CD>>` to an `IntStream` whose values are the number of tracks in a `CD` contained in the list. The resulting `Stream<IntStream>` from the mapper function is flattened into an `IntStream` by the `flatMapToInt()` intermediate operation, thus transforming the initial `Stream<List<CD>>` into an `IntStream`. The terminal operation `sum()` sums the values in this `IntStream` (p. 973).

Click here to view code image

```
List<CD> cdList1 = List.of(CD.cd0, CD.cd1);
List<CD> cdList2 = List.of(CD.cd2, CD.cd3, CD.cd4);
int totalNumOfTracks =
    Stream.of(cdList1, cdList2)                       // Stream<List<CD>>
          .flatMapToInt(                              // (1)
              lst -> lst.stream()                     // Stream<CD>
                        .mapToInt(CD::noOfTracks))    // IntStream
                                                      // Stream<IntStream>,
                                                      //   flattened to IntStr
          .sum();                                     // 42
```

Mapping a Numeric Stream to an Object Stream

The `mapToObj()` method defined by the numeric stream interfaces transforms a numeric stream to an object stream of type `R`, and the `boxed()` method transforms a numeric stream to an object stream of its wrapper class.

The query below prints the squares of numbers in a given closed range, where the number and its square are stored as a pair in a list of size 2. The `mapToObj()` intermediate operation at (2) transforms an `IntStream` created at (1) to a `Stream<List<Integer>>`. Each list in the result stream is then printed by the `forEach()` terminal operation.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                            // (1) IntStream
         .mapToObj(n -> List.of(n, n*n))               // (2) Stream<List<Intege
         .forEach(p -> System.out.print(p + " "));     // [1, 1] [2, 4] [3, 9]
```

The query above can also be expressed as shown below. The `boxed()` intermediate operation transforms the `IntStream` at (3) into a `Stream<Integer>` at (4); in other words, each `int` value is boxed into an `Integer` which is then mapped by the `map()` operation at (5) to a `List<Integer>`, resulting in a `Stream<List<Integer>>` as before. The compiler will issue an error if the `boxed()` operation is omitted at (4), as the `map()` operation at (5) will be invoked on an `IntStream`, expecting an `IntUnaryFunction`, which is not the case.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                            // (3) IntStream
         .boxed()                                      // (4) Stream<Integer>
         .map(n -> List.of(n, n*n))                    // (5) Stream<List<Intege
         .forEach(p -> System.out.print(p + " "));     // [1, 1] [2, 4] [3, 9]
```

The examples above show that the `IntStream.mapToObj()` method is equivalent to the `IntStream.boxed()` method followed by the `Stream.map()` method.

The `mapToObj()` method, in conjunction with a range of `int` values, can be used to create sublists and subarrays. The query below creates a sublist of `CD` titles based on a closed range whose values are used as an index in the `CD` list.

Click here to view code image

```
List<String> subListTitles = IntStream
    .rangeClosed(2, 3)                          // IntStream
    .mapToObj(i -> CD.cdList.get(i).title())    // Stream<String>
    .toList();                                  // [Lambda Dancing, Keep on Eras
```

Mapping between Numeric Streams

In contrast to the methods in the `Stream<T>` interface, the `map()` and the `flatMap()` methods of the numeric stream interfaces transform a numeric stream to a numeric stream of the *same* primitive type; that is, they do *not* change the type of the numeric stream.

The `map()` operation in the stream pipeline below does not change the type of the initial `IntStream`.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                           // IntStream
         .map(i -> i * i)                             // IntStream
         .forEach(n -> System.out.printf("%d ", n));  // 1 4 9
```

The `flatMap()` operation in the stream pipeline below also does not change the type of the initial stream. Each `IntStream` created by the mapper function is flattened, resulting in a single `IntStream`.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                           // IntStream
         .flatMap(i -> IntStream.rangeClosed(1, 4))   // IntStream
         .forEach(n -> System.out.printf("%d ", n));  // 1 2 3 4 1 2 3 4 1 2 3
```

Analogous to the methods in the `Stream<T>` interface, the `mapTo`*NumType*`()` methods in the numeric stream interfaces transform a numeric stream to a stream of the designated numeric type, where *`NumType`* is either `Int`, `Long`, or `Double`.

The `mapToDouble()` operation in the stream pipeline below transforms the initial `IntStream` into a `DoubleStream`.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                           // IntStream
         .mapToDouble(i -> Math.sqrt(i))              // DoubleStream
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

The methods `asLongStream()` and `asDoubleStream()` in the `IntStream` interface transform an `IntStream` to a `LongStream` and a `DoubleStream`, respectively. Similarly, the method `asDoubleStream()` in the `LongStream` interface transforms a `LongStream` to a `DoubleStream`.

The `asDoubleStream()` operation in the stream pipeline below transforms the initial `IntStream` into a `DoubleStream`. Note how the range of `int` values is thereby transformed to a range of `double` values by the `asDoubleStream()` operation.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                                 // IntStream
         .asDoubleStream()                                  // DoubleStream
         .map(d -> Math.sqrt(d))                            // DoubleStream
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

In the stream pipeline below, the `int` values in the `IntStream` are first boxed into `Integer`s. In other words, the initial `IntStream` is transformed into an *object stream*, `Stream<Integer>`. The `map()` operation transforms the `Stream<Integer>` into a `Stream<Double>`. In contrast to using the `asDoubleStream()` in the stream pipeline above, note the boxing/unboxing that occurs in the stream pipeline below in the evaluation of the `Math.sqrt()` method, as this method accepts a `double` as a parameter and returns a `double` value.

Click here to view code image

```
IntStream.rangeClosed(1, 3)                                 // IntStream
         .boxed()                                           // Stream<Integer>
         .map(n -> Math.sqrt(n))                            // Stream<Double>
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

Summary of Intermediate Stream Operations

Table 16.3 summarizes selected aspects of the intermediate operations.

Table 16.3 *Selected Aspects of Intermediate Stream Operations*

Intermediate operation	Stateful/Stateless	Can change stream size	Can change stream type	Encounter order
`distinct` (p. 915)	*Stateful*	*Yes*	*No*	*Unchanged*
`dropWhile` (p. 913)	*Stateful*	*Yes*	*No*	*Unchanged*
`filter` (p. 910)	*Stateless*	*Yes*	*No*	*Unchanged*
`flatMap` (p. 921)	*Stateless*	*Yes*	*Yes*	*Not guaranteed*
`limit` (p. 917)	*Stateful, short-circuited*	*Yes*	*No*	*Unchanged*
`map` (p. 921)	*Stateless*	*No*	*Yes*	*Not guaranteed*
`mapMulti` (p. 927)	*Stateless*	*Yes*	*Yes*	*Not guaranteed*
`parallel` (p. 933)	*Stateless*	*No*	*No*	*Unchanged*
`peek` (p. 920)	*Stateless*	*No*	*No*	*Unchanged*
`sequential` (p. 933)	*Stateless*	*No*	*No*	*Unchanged*
`skip` (p. 915)	*Stateful*	*Yes*	*No*	*Unchanged*
`sorted` (p. 929)	*Stateful*	*No*	*No*	*Ordered*
`takeWhile` (p. 913)	*Stateful, short-circuited*	*Yes*	*No*	*Unchanged*
`unordered` (p. 932)	*Stateless*	*No*	*No*	*Not guaranteed*

The intermediate operations of the `Stream<T>` interface (including those inherited from its superinterface `BaseStream<T,Stream<T>>`) are summarized in Table 16.4. The type parameter declarations have been simplified, where any bounds `<? super T>` or `<?`

`extends T>` have been replaced by `<T>`, without impacting the intent of a method. A reference is provided to each method in the first column. Any type parameter and return type declared by these methods are shown in column two.

The last column in Table 16.4 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior of an operation. For example, the `filter()` method returns a stream whose elements satisfy a given predicate. This predicate is defined by the functional interface `Predicate<T>` that is implemented by a lambda expression or a method reference, and applied to each element in the stream.

The interfaces `IntStream`, `LongStream`, and `DoubleStream` also define analogous methods to those shown in Table 16.4, except for the `flatMapTo`*`NumType`*`()` methods, where *`NumType`* is either `Int`, `Long`, or `Double`. A summary of additional methods defined by these numeric stream interfaces can be found in Table 16.2.

Table 16.4 *Intermediate Stream Operations*

Method name	Any type parameter	Return type	Functional interface parameters
`distinct (p. 915)`		`Stream<T>`	`()`
`dropWhile (p. 913)`		`Stream<T>`	`(Predicate<T> predicate)`
`filter (p. 910)`		`Stream<T>`	`(Predicate<T> predicate)`
`flatMap (p. 921)`	`<R>`	`Stream<R>`	`(Function<T,Stream<R>> mapper)`
`flatMapToDouble (p. 921)`		`DoubleStream`	`(Function<T,DoubleStream> mapper)`
`flatMapToInt (p. 921)`		`IntStream`	`(Function<T,IntStream> mapper)`
`flatMapToLong (p. 921)`		`LongStream`	`(Function<T,LongStream> mapper)`
`limit (p. 917)`		`Stream<T>`	`(long maxSize)`
`map (p. 921)`	`<R>`	`Stream<R>`	`(Function<T,R> mapper)`
`mapMulti (p. 927)`	`<R>`	`Stream<R>`	`(BiConsumer<T,Consumer<R>> mapper)`
`mapToDouble (p. 921)`		`DoubleStream`	`(ToDoubleFunction<T> mapper)`
`mapToInt (p. 921)`		`IntStream`	`(ToIntFunction<T> mapper)`
`mapToLong (p. 921)`		`LongStream`	`(ToLongFunction<T> mapper)`
`parallel (p. 933)`		`Stream<T>`	`()`
`peek (p. 920)`		`Stream<T>`	`(Consumer<T> action)`
`sequential (p. 933)`		`Stream<T>`	`()`
`skip (p. 915)`		`Stream<T>`	`(long n)`
`sorted (p. 929)`		`Stream<T>`	`()`
`sorted (p. 929)`		`Stream<T>`	`(Comparator<T> cmp)`
`takeWhile (p. 913)`		`Stream<T>`	`(Predicate<T> predicate)`
`unordered (p. 932)`		`Stream<T>`	`()`

16.6 The `Optional` Class

When a method returns a `null` value, it is not always clear whether the `null` value represents a valid value or the *absence of a value*. Methods that can return `null` values invariably force their callers to check the returned value explicitly in order to avoid a `NullPointerException` before using the returned value. For example, method chaining, which we have seen for composing stream pipelines, becomes cumbersome if each method call must be checked to see whether it returns a `null` value before calling the next method, resulting in a cascade of conditional statements.

The concept of an `Optional` object allows the absence of a value to be handled in a systematic way, making the code robust by enforcing that a consumer of an `Optional` must also deal with the case when the value is absent. Taking full advantage of `Optional` wrappers requires using them the right way, primarily to handle situations where the value returned by a method is absent.

The generic class `Optional<T>` provides a wrapper that represents either the presence or absence of a non-`null` value of type `T`. In other words, the wrapper either contains a non-`null` value of type `T` or no value at all.

Example 16.8 illustrates using objects of the `Optional<T>` class.

Declaring and Returning an `Optional`

Example 16.8 illustrates declaring and returning an `Optional`. A book is represented by the `Book` class that has an optional blurb of type `String`; that is, a book may or may not have a blurb. The `Optional<T>` class is parameterized with the type `String` in the declaration, and so is the return type of the method that returns the optional blurb.

Click here to view code image

```
class Book {
  private Optional<String> optBlurb;

  public Optional<String> getOptBlurb() { return optBlurb; }

  //...
}
```

Creating an `Optional`

The `Optional<T>` class models the absence of a value by a special singleton returned by the `Optional.empty()` method. In contrast to the `null` value, this singleton is a viable `Optional` object on which methods of the `Optional` class can be invoked without a `NullPointerException` being thrown.

> Click here to view code image
>
> ```
> static <T> Optional<T> empty()
> static <T> Optional<T> of(T nonNullValue)
> static <T> Optional<T> ofNullable(T value)
> ```
>
> The `empty()` method returns an empty `Optional` instance; that is, it indicates *the absence of a value*.
>
> The `of()` method returns an `Optional` with the specified value, if this value is non-`null`. Otherwise, a `NullPointerException` is thrown.
>
> The `ofNullable()` method returns an `Optional` with the specified value, if this value is non-`null`. Otherwise, it returns an empty `Optional`.

The static `Optional.of()` factory method creates an `Optional` that encapsulates the non-`null` argument specified in the method call, as in the first declaration below. However, if the argument is a `null` value, a `NullPointerException` is thrown at runtime, as in the second declaration.

Click here to view code image

```
Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
Optional<String> xblurb = Optional.of(null);   // NullPointerException
```

The static `Optional.ofNullable()` factory method also creates an `Optional` that encapsulates the non-`null` argument specified in the method call, as in the first declaration below. However, if the argument is a `null` value, the method returns an *empty* `Optional`, as in the second declaration—which is effectively the same as the third declaration below.

Click here to view code image

```
Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
Optional<String> noBlurb2 = Optional.ofNullable(null);   // Optional.empty()
Optional<String> noBlurb3 = Optional.empty();
```

The blurbs above are used to initialize two `Book` objects (`book0`, `book1`) in Example 16.8. These `Book` objects with optional blurbs will be used to illustrate how to use `Optional` objects.

Example 16.8 *Using Optionals*

Click here to view code image

```
// File: OptionalUsage.java
import java.util.Optional;

// A book can have an optional blurb.
class Book {
  private String bookName;
  private Optional<String> optBlurb;

  public String getBookName() { return bookName; }
  public Optional<String> getOptBlurb() { return optBlurb; }
  public Book(String bookName, Optional<String> optBlurb) {
    this.bookName = bookName;
    this.optBlurb = optBlurb;
  }
}

// A course can have an optional book.
class Course {
  private Optional<Book> optBook;
  public Optional<Book> getOptBook() { return optBook; }
  public Course(Optional<Book> optBook) { this.optBook = optBook; }
}

public class OptionalUsage {
  public static void main(String[] args) {

    // Creating an Optional:
    Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
    //Optional<String> xblurb = Optional.of(null);   // NullPointerException
    Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
    Optional<String> noBlurb2 = Optional.ofNullable(null);   // Optional.empty
    Optional<String> noBlurb3 = Optional.empty();

    // Create some books:
    Book book0 = new Book("Embarrassing Exceptions", blurb0);
    Book book1 = new Book("Dancing Lambdas", noBlurb2);        // No blurb.

    // Querying an Optional:
    if (book0.getOptBlurb().isPresent()) {
      System.out.println(book0.getOptBlurb().get());//Java Programmers tell al
    }

    book0.getOptBlurb()
         .ifPresent(System.out::println);           //Java Programmers tell al

//  System.out.println(book1.getOptBlurb().get());   // NoSuchElementException

    String blurb = book0.getOptBlurb()
                        .orElse("No blurb");     // "Java Programmers tell all
    System.out.println(blurb);

    blurb = book1.getOptBlurb().orElse("No blurb");          // "No blurb"
    System.out.println(blurb);

    blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
    System.out.println(blurb);

    //blurb = book1.getOptBlurb()                             // RuntimeExcepti
    //              .orElseThrow(() -> new RuntimeException("No blurb"));
  }
}
```

Output from the program:

```
Java Programmers tell all!
Java Programmers tell all!
Java Programmers tell all!
No blurb
No blurb
```

Querying an `Optional`

The presence of a value in an `Optional` can be determined by the `isPresent()` method, and the value can be obtained by calling the `get()` method—which is not much better than checking explicitly for the `null` value, but as we shall see, other methods in the `Optional` class alleviate this drudgery. The `get()` method throws a `NoSuchElementException` if there is no value in the `Optional`.

Click here to view code image

```
if (book0.getOptBlurb().isPresent()) {
  System.out.println(book0.getOptBlurb().get());    // Java Programmers tell a
}
System.out.println(book1.getOptBlurb().get());      // NoSuchElementException
```

The idiom of determining the presence of a value and then handling the value is combined by the `ifPresent()` method that accepts a `Consumer` to handle the value if one is present. The `ifPresent()` method does nothing if there is no value present in the `Optional`.

Click here to view code image

```
book0.getOptBlurb().ifPresent(System.out::println); //Java Programmers tell al
```

```
T get()
```

If a value is present in this `Optional`, the method returns that value; otherwise, it throws a `NoSuchElementException`.

Click here to view code image

```
boolean isPresent()
void ifPresent(Consumer<? super T> consumer)
```

The first method returns `true` if there is a value present in this `Optional`; otherwise, it returns `false`.

If a value is present in this `Optional`, the second method invokes the specified `consumer` with the value; otherwise, it

> does nothing.
>
> Click here to view code image
>
> ```
> T orElse(T other)
> T orElseGet(Supplier<? extends T> other)
> <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
>                         throws X extends Throwable
> ```
>
> If a value is present in this `Optional`, all three methods return this value.
>
> They differ in their action when a value is not present in this `Optional`. The first method returns the `other` value. The second method invokes the specified supplier `other` and returns its result. The third method invokes the specified supplier `exceptionSupplier` and throws the exception created by this supplier.
>
> Note that the type of argument in the first two methods must be compatible with the parameterized type of the `Optional` on which the method is invoked, or the compiler will issue an error.

Often, a default value should be supplied when an `Optional` does *not* contain a value. The `orElse()` method returns the value in the `Optional` if one is present; otherwise, it returns the value given by the argument specified in the method call.

The `orElse()` method in the statement below returns the blurb in the book referenced by the reference `book0`, as this book has a blurb.

Click here to view code image

```
String blurb = book0.getOptBlurb()
                    .orElse("No blurb");          // "Java Programmers tell a
```

The book referenced by the reference `book1` has no blurb. Therefore, the `orElse()` method invoked on the optional blurb returns the argument in the method.

Click here to view code image

```
blurb = book1.getOptBlurb().orElse("No blurb");    // "No blurb"
```

For an `Optional` with a value, the `orElseGet()` method returns the value in the `Optional`. The `orElseGet()` method in the statement below returns the object supplied by the `Supplier` specified as an argument, since the book has no blurb.

Click here to view code image

```
blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
```

For an `Optional` with a value, the `orElseThrow()` method also returns the value in the `Optional`. The `orElseThrow()` method in the statement below throws the exception created by the `Supplier` specified as an argument, since the book has no blurb.

Click here to view code image

```
blurb = book1.getOptBlurb()                          // RuntimeException
             .orElseThrow(() -> new RuntimeException("No blurb"));
```

Numeric Optional Classes

An instance of the generic `Optional<T>` class can only encapsulate an object. To deal with optional numeric values, the `java.util` package also defines the following non-generic classes that can encapsulate primitive numeric values: `OptionalInt`, `OptionalLong`, and `OptionalDouble`. For example, an `OptionalInt` object encapsulates an `int` value.

The numeric optional classes provide methods analogous to the static factory methods of the `Optional` class to create a numeric optional from a numeric value and methods to query a numeric optional. The `filter()`, `map()`, and `flatMap()` methods are *not* defined for the numeric optional classes.

The following methods are defined in the `OptionalInt`, `OptionalLong`, and `Optional-Double` classes in the `java.util` package. In the methods below, *`NumType`* is `Int`, `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`.

> Click here to view code image
>
> ```
> static OptionalNumType empty()
> static OptionalNumType of(numtype value)
> ```
>
> These two methods return an empty `Optional`*NumType* instance and an `Optional-`*NumType* with the specified `value`, respectively.
>
> Click here to view code image
>
> ```
> boolean isPresent()
> void    ifPresent(NumTypeConsumer consumer)
> ```
>
> If there is a value present in this `Optional`, the first method returns `true`. Otherwise, it returns `false`.

If a value is present in this `Optional`, the specified consumer is invoked on the value. Otherwise, it does nothing.

```
numtype getAsNumType()
```

If a value is present in this `Optional`*NumType*, the method returns the value. Otherwise, it throws a `NoSuchElementException`.

Click here to view code image

```
numtype orElse(numtype other)
numtype orElseGet(NumTypeSupplier other)
<X extends Throwable> numtype orElseThrow(Supplier<X> exceptionSupplier)
                                          throws X extends Throwable
```

If a value is present in this `Optional`*NumType*, all three methods return this value.

They differ in their action when there is no value present in this numeric optional. The first method returns the specified `other` value. The second method invokes the specified `other` supplier and returns the result. The third method throws an exception that is created by the specified supplier.

Example 16.9 illustrates using numeric optional values. A recipe has an optional number of calories that are modeled using an `OptionalInt` that can encapsulate an `int` value. Declaring, creating, and querying `OptionalInt` objects is analogous to that for `Optional` objects.

Example 16.9 *Using Numerical Optionals*

Click here to view code image

```
// File: NumericOptionalUsage.java
import java.util.OptionalInt;

class Recipe {
  private String recipeName;
  private OptionalInt calories;     // Optional number of calories.

  public String getRecipeName() { return recipeName; }
  public OptionalInt getCalories() { return calories; }

  public Recipe(String recipeName, OptionalInt calories) {
    this.recipeName = recipeName;
    this.calories = calories;
  }
}

public final class NumericOptionalUsage {
  public static void main(String[] args) {
    // Creating an OptionalInt:
      OptionalInt optNOC0 = OptionalInt.of(3500);
      OptionalInt optNOC1 = OptionalInt.empty();

      // Creating recipes with optional number of calories:
      Recipe recipe0 = new Recipe("Mahi-mahi", optNOC0);
      Recipe recipe1 = new Recipe("Loco moco", optNOC1);

      // Querying an Optional:
      //  System.out.println(recipe1.getCalories()
      //                            .getAsInt());          // NoSuchElementExcep
      System.out.println((recipe1.getCalories().isPresent()
          ? recipe1.getCalories().getAsInt()
              : "Unknown calories."));       // Unknown calories.

      recipe0.getCalories().ifPresent(s -> System.out.println(s + " calories."))
      System.out.println(recipe0.getCalories().orElse(0) + " calories.");
      System.out.println(recipe1.getCalories().orElseGet(() -> 0) + " calories."
      //  int noc = recipe1.getCalories()                    // RuntimeException
      //              .orElseThrow(() -> new RuntimeException("Unknown calories.
    }
  }
```

Output from the program:

```
Unknown calories.
3500 calories.
3500 calories.
0 calories.
```

16.7 Terminal Stream Operations

A stream pipeline does not execute until a terminal operation is invoked on it; that is, a stream pipeline does not start to process the stream elements until a terminal operation is initiated. A terminal operation is said to be *eager* as it executes immediately when invoked—as opposed to an intermediate operation which is *lazy*. Invoking the terminal operation results in the intermediate operations of the stream pipeline to be executed. Understandably, a terminal operation is specified as the last operation in a stream pipeline, and there can only be one such operation in a stream pipeline. A terminal operation never returns a stream, which is always done by an intermediate operation. Once the terminal operation completes, the stream is consumed and cannot be reused.

Terminal operations can be broadly grouped into three groups:

- *Operations with side effects*

  The Stream API provides two terminal operations, `forEach()` and `forEachOrdered()`, that are designed to allow side effects on stream elements (p. 948). These terminal operations do not return a value. They allow a

`Consumer` action, specified as an argument, to be applied to every element, as they are consumed from the stream pipeline—for example, to print each element in the stream.

- *Searching operations*

  These operations perform a search operation to determine a match or find an element as explained below.

  All search operations are *short-circuit operations*; that is, the operation can terminate once the result is determined, whether or not all elements in the stream have been considered.

  Search operations can be further classified into two subgroups:

  ❍ *Matching operations*

  The three terminal operations `anyMatch()`, `allMatch()`, and `noneMatch()` determine whether stream elements match a given `Predicate` specified as an argument to the method (p. 949). As expected, these operations return a `boolean` value to indicate whether the match was successful or not.

  ❍ *Finding operations*

  The two terminal operations `findAny()` and `findFirst()` find any element and the first element in a stream, respectively, if such an element is available (p. 952). As the stream might be empty and such an element might not exist, these operations return an `Optional`.

- *Reduction operations*

  A *reduction* operation computes a result from combining the stream elements by successively applying a combining function; that is, the stream elements are *reduced* to a result value. Examples of reductions are computing the sum or average of numeric values in a numeric stream, and accumulating stream elements into a collection.

  We distinguish between two kinds of reductions:

  ❍ *Functional reduction*

  A terminal operation is a functional reduction on the elements of a stream if it reduces the elements to a *single immutable value* which is then returned by the operation.

The overloaded `reduce()` method provided by the Stream API can be used to implement customized functional reductions (p. 955), whereas the terminal operations `count()`, `min()`, and `max()` implement specialized functional reductions (p. 953).

Functional reductions on numeric streams are discussed later in this section (p. 972).

❍ *Mutable reduction*

A terminal operation performs a *mutable reduction* on the elements of a stream if it uses *a mutable container*—for example, a list, a set, or a map—to accumulate values as it processes the stream elements. The operation returns the mutable container as the result of the operation.

The Stream API provides two overloaded `collect()` methods that perform mutable reduction (p. 964). One overloaded `collect()` method can be used to implement customized mutable reductions by specifying the functions (*supplier*, *accumulator*, *combiner*) required to perform such a reduction. A second `collect()` method accepts a `Collector` that is used to perform a mutable reduction. A *collector* encapsulates the functions required for performing a mutable reduction. The Stream API provides built-in collectors that allow various containers to be used for performing mutable reductions (p. 978). When a terminal operation performs a mutable reduction using a specific container, it is said to *collect to* this container.

The `toArray()` method implements a specialized mutable reduction that returns an array with the accumulated values (p. 971); that is, the method collects to an array.

Consumer Action on Stream Elements

We have already used both the `forEach()` and `forEachOrdered()` terminal operations to print elements when the pipeline is executed. These operations allow side effects on stream elements.

The `forEach()` method is defined for both streams and collections. In the case of collections, the method iterates over all the elements in the collection, whereas it is a terminal operation on streams.

Since these terminal operations perform an action on each element, the input stream to the operation must be *finite* in order for the operation to terminate.

Counterparts to the `forEach()` and `forEachOrdered()` methods for the primitive numeric types are also defined by the numeric stream interfaces.

> Click here to view code image
>
> ```
> void forEach(Consumer<? super T> action)
> ```
>
> This terminal operation performs an action on each element of this stream. This method should not be relied upon to produce deterministic results, as the order in which the elements are processed is not guaranteed.
>
> Click here to view code image
>
> ```
> void forEachOrdered(Consumer<? super T> action)
> ```
>
> This terminal operation performs an action on each element of this stream, but in the encounter order of the stream if the stream has one.

The difference in behavior of the `forEach()` and `forEachOrdered()` terminal operations is that the `forEach()` method does not guarantee to respect the encounter order, whereas the `forEachOrdered()` method always does, if there is one.

Each operation is applied to both an ordered sequential stream and an ordered parallel stream to print CD titles with the help of the consumer `printStr`:

Click here to view code image

```
Consumer<String> printStr = str -> System.out.print(str + "|");

CD.cdList.stream().map(CD::title).forEach(printStr);                 // (1a)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|


CD.cdList.stream().parallel().map(CD::title).forEach(printStr);      // (1b)
//Lambda Dancing|Hot Generics|Keep on Erasing|Java Jam|Java Jive|
```

The behavior of the `forEach()` operation is nondeterministic, as seen at (1a) and (1b). The output from (1a) and (1b) shows that the `forEach()` operation respects the encounter order for an ordered sequential stream, but not necessarily for an ordered parallel stream. Respecting the encounter order for an ordered parallel stream would incur overhead that would impact performance, and is therefore ignored.

On the other hand, the `forEachOrdered()` operation always respects the encounter order in both cases, as seen below from the output at (2a) and (2b). However, it is important to note that, in the case of the ordered parallel stream, the terminal action on the elements can be executed in different threads, but guarantees that the action is applied to the elements in encounter order.

Click here to view code image

```
CD.cdList.stream().map(CD::title).forEachOrdered(printStr);              // (2
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|

CD.cdList.stream().parallel().map(CD::title).forEachOrdered(printStr);   // (2
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
```

The discussion above also applies when the `forEach()` and `forEachOrdered()` terminal operations are invoked on numeric streams. The nondeterministic behavior of the `forEach()` terminal operation for `int` streams is illustrated below. The terminal operation on the sequential `int` stream at (3a) seems to respect the encounter order, but should not be relied upon. The terminal operation on the parallel `int` stream at (3b) can give different results for different runs.

Click here to view code image

```
IntConsumer printInt = n -> out.print(n + "|");

IntStream.of(2018, 2019, 2020, 2021, 2022).forEach(printInt);            // (3
//2018|2019|2020|2021|2022|

IntStream.of(2018, 2019, 2020, 2021, 2022).parallel().forEach(printInt); // (3
//2020|2019|2018|2021|2022|
```

Matching Elements

The match operations determine whether any, all, or none of the stream elements satisfy a given `Predicate`. These operations are not reductions, as they do not always consider all elements in the stream in order to return a result.

Analogous match operations are also provided by the numeric stream interfaces.

Click here to view code image

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

These three terminal operations determine whether *any*, *all*, or

> *no* elements of this stream match the specified predicate, respectively.
>
> The methods may not evaluate the predicate on all elements if it is not necessary for determining the result; that is, they are *short-circuit* operations.
>
> If the stream is empty, the predicate is *not* evaluated.
>
> The `anyMatch()` method returns `false` if the stream is empty.
>
> The `allMatch()` and `noneMatch()` methods return `true` if the stream is empty.
>
> There is no guarantee that these operations will terminate if applied to an infinite stream.

The queries at (1), (2), and (3) below determine whether any, all, or no CDs are jazz music CDs, respectively. At (1), the execution of the pipeline terminates as soon as any jazz music CD is found—the value `true` is returned. At (2), the execution of the pipeline terminates as soon as a non-jazz music CD is found—the value `false` is returned. At (3), the execution of the pipeline terminates as soon as a jazz music CD is found—the value `false` is returned.

Click here to view code image

```
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz);   // (1) true
boolean allJazzCds = CD.cdList.stream().allMatch(CD::isJazz);  // (2) false
boolean noJazzCds = CD.cdList.stream().noneMatch(CD::isJazz);  // (3) false
```

Given the following predicates:

Click here to view code image

```
Predicate<CD> eq2015 = cd -> cd.year().compareTo(Year.of(2015)) == 0;
Predicate<CD> gt2015 = cd -> cd.year().compareTo(Year.of(2015)) > 0;
```

The query at (4) determines that no CDs were released in 2015. The queries at (5) and (6) are equivalent. If all CDs were released after 2015, then none were released in or before 2015 (negation of the predicate `gt2015`).

Click here to view code image

```
boolean noneEQ2015 = CD.cdList.stream().noneMatch(eq2015);     // (4) true
boolean allGT2015 = CD.cdList.stream().allMatch(gt2015);       // (5) true
boolean noneNotGT2015 = CD.cdList.stream().noneMatch(gt2015.negate()); // (6)
```

The code below uses the `anyMatch()` method on an `int` stream to determine whether any year is a leap year.

Click here to view code image

```
IntStream yrStream = IntStream.of(2018, 2019, 2020);
IntPredicate isLeapYear = yr -> Year.of(yr).isLeap();
boolean anyLeapYear = yrStream.anyMatch(isLeapYear);
out.println("Any leap year: " + anyLeapYear);   // true
```

Example 16.10 illustrates using the `allMatch()` operation to determine whether a square matrix—that is, a two-dimensional array with an equal number of columns as rows—is an *identity matrix*. In such a matrix, all elements on the main diagonal have the value 1 and all other elements have the value 0. The methods `isIdentityMatrixLoops()` and `isIdentityMatrixStreams()` at (1) and (2) implement this test in different ways.

The method `isIdentityMatrixLoops()` at (1) uses nested loops. The outer loop processes the rows, whereas the inner loop tests that each row has the correct values. The outer loop is a labeled loop in order to break out of the inner loop if an element in a row does not have the correct value—effectively achieving short-circuit execution.

The method `isIdentityMatrixStreams()` at (2) uses nested numeric streams, where the outer stream processes the rows and the inner stream processes the elements in a row. The `allMatch()` method at (4) in the inner stream pipeline determines that all elements in a row have the correct value. It short-circuits the execution of the inner stream if that is not the case. The `allMatch()` method at (3) in the outer stream pipeline also short-circuits its execution if its predicate to process a row returns the value `false`. The stream-based implementation for the identity matrix test expresses the logic more clearly and naturally than the loop-based version.

Example 16.10 *Identity Matrix Test*

Click here to view code image

```
import static java.lang.System.out;

import java.util.Arrays;
import java.util.stream.IntStream;

public class IdentityMatrixTest {
  public static void main(String[] args) {
    // Matrices to test:
    int[][] sqMatrix1 = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
    int[][] sqMatrix2 = { {1, 1}, {1, 1} };
    isIdentityMatrixLoops(sqMatrix1);
    isIdentityMatrixLoops(sqMatrix2);
    isIdentityMatrixStreams(sqMatrix1);
    isIdentityMatrixStreams(sqMatrix2);
  }

  private static void isIdentityMatrixLoops(int[][] sqMatrix) {          // (
    boolean isCorrectValue = false;
    outerLoop:
    for (int i = 0; i < sqMatrix.length; ++i) {
      for (int j = 0; j < sqMatrix[i].length; ++j) {
        isCorrectValue = j == i ? sqMatrix[i][i] == 1
                                : sqMatrix[i][j] == 0;
        if (!isCorrectValue) break outerLoop;
      }
    }
    out.println(Arrays.deepToString(sqMatrix)
        + (isCorrectValue ? " is ": " is not ") + "an identity matrix.");
  }

  private static void isIdentityMatrixStreams(int[][] sqMatrix) {       // (2
    boolean isCorrectValue =
        IntStream.range(0, sqMatrix.length)
                 .allMatch(i -> IntStream.range(0, sqMatrix[i].length)   // (3
                                         .allMatch(j -> j == i           // (4
                                                   ? sqMatrix[i][i] == 1
                                                   : sqMatrix[i][j] == 0));
    out.println(Arrays.deepToString(sqMatrix)
        + (isCorrectValue ? " is ": " is not ") + "an identity matrix.");
  }
}
```

Output from the program:

Click here to view code image

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
```

Finding the First or Any Element

The `findFirst()` method can be used to find the first element that is available in the stream. This method respects the encounter order, if the stream has one. It always produces a stable result; that is, it will produce the same result on identical pipelines based on the same stream source. In contrast, the behavior of the `findAny()` method is nondeterministic. Counterparts to these methods are also defined by the numeric stream interfaces.

> `Optional<T> findFirst()`
>
> This terminal operation returns an `Optional` describing the *first* element of this stream, or an empty `Optional` if the stream is empty.
>
> This method may return any element if this stream does not

> have any encounter order.
>
> It is a *short-circuit* operation, as it will terminate the execution of the stream pipeline as soon as the first element is found.
>
> This method throws a `NullPointerException` if the element selected is `null`.
>
> ```
> Optional<T> findAny()
> ```
>
> This terminal operation returns an `Optional` describing *some* element of the stream, or an empty `Optional` if the stream is empty. This operation has nondeterministic behavior.
>
> It is a *short-circuit* operation, as it will terminate the execution of the stream pipeline as soon as any element is found.

In the code below, the encounter order of the stream is the positional order of the elements in the list. The first element returned by the `findFirst()` method at (1) is the first element in the CD list.

Click here to view code image

```
Optional<CD> firstCD1 = CD.cdList.stream().findFirst();          // (1)
out.println(firstCD1.map(CD::title).orElse("No first CD."));     // (2) Java Ji
```

Since such an element might not exist—for example, the stream might be empty— the method returns an `Optional<T>` object. At (2), the `Optional<CD>` object returned by the `findFirst()` method is mapped to an `Optional<String>` object that encapsulates the title of the CD. The `orElse()` method on this `Optional<String>` object returns the CD title or the argument string if there is no such CD.

If the encounter order is not of consequence, the `findAny()` method can be used, as it is nondeterministic—that is, it does not guarantee the same result on the same stream source. On the other hand, it provides maximal performance on parallel streams. At (3) below, the `findAny()` method is free to return any element from the parallel stream. It should not come as a surprise if the element returned is not the first element in the list.

Click here to view code image

```
Optional<CD> anyCD2 = CD.cdList.stream().parallel().findAny();  // (3)
out.println(anyCD2.map(CD::title).orElse("No CD."));            // Lambda Danc
```

The match methods only determine whether any elements satisfy a `Predicate`, as seen at (5) below. Typically, a find terminal operation is used to find the first element made available to the terminal operation after processing by the intermediate operations in the stream pipeline. At (6), the `filter()` operation will filter the jazz music CDs from the stream. However, the `findAny()` operation will return the first jazz music CD that is filtered and then short-circuit the execution.

Click here to view code image

```
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz);              // (
out.println("Any Jazz CD: " + anyJazzCD);   // Any Jazz CD: true

Optional<CD> optJazzCD = CD.cdList.stream().filter(CD::isJazz).findAny(); // (
optJazzCD.ifPresent(out::println);          // <Jaav, "Java Jam", 6, 2017, JAZ
```

The code below uses the `findAny()` method on an `IntStream` to find whether any number is divisible by 7.

Click here to view code image

```
IntStream numStream = IntStream.of(50, 55, 65, 70, 75, 77);
OptionalInt intOpt = numStream.filter(n -> n % 7 == 0).findAny();
intOpt.ifPresent(System.out::println);      // 70
```

The find operations are guaranteed to terminate when applied to a finite, albeit empty, stream. However, for an infinite stream in a pipeline, at least one element must be made available to the find operation in order for the operation to terminate. If the elements of an initial infinite stream are all discarded by the intermediate operations, the find operation will not terminate, as in the following pipeline:

Click here to view code image

```
Stream.generate(() -> 1).filter(n -> n == 0).findAny();       // Never termina
```

Counting Elements

The `count()` operation performs a functional reduction on the elements of a stream, as each element contributes to the count which is the single immutable value returned by the operation. The `count()` operation reports the number of elements that are made available to it, which is not necessarily the same as the number of elements in the initial stream, as elements might be discarded by the intermediate operations.

The code below finds the total number of CDs in the streams, and how many of these CDs are jazz music CDs.

Click here to view code image

```
long numOfCDS = CD.cdList.stream().count();                          // 5
long numOfJazzCDs = CD.cdList.stream().filter(CD::isJazz).count(); // 3
```

The `count()` method is also defined for the numeric streams. Below it is used on an `IntStream` to find how many numbers between 1 and 100 are divisible by 7.

Click here to view code image

```
IntStream numStream = IntStream.rangeClosed(1, 100);
long divBy7 = numStream.filter(n -> n % 7 == 0).count();            // 14
```

> `long count()`
>
> This terminal operation returns the count of elements in this stream—that is, the length of this stream.
>
> This operation is a special case of a functional reduction.
>
> The operation does not terminate when applied to an infinite stream.

Finding Min and Max Elements

The `min()` and `max()` operations are functional reductions, as they consider all elements of the stream and return a single value. They should only be applied to a finite stream, as they will not terminate on an infinite stream. These methods are also defined by the numeric stream interfaces for the numeric types, but without the specification of a comparator.

> Click here to view code image
>
> ```
> Optional<T> min(Comparator<? super T> cmp)
> Optional<T> max(Comparator<? super T> cmp)
> ```
>
> These terminal operations return an `Optional` with the minimum or maximum element of this stream according to the provided `Comparator`, respectively, or an empty `Optional` if this stream is empty. It throws a `NullPointerException` if the minimum element is `null`.
>
> These operations are a special case of a functional reduction.
>
> These operations do not terminate when applied to an infinite stream.

Both methods return an `Optional`, as the minimum and maximum elements might not exist—for example, if the stream is empty. The code below finds the minimum and maximum elements in a stream of CDs, according to their natural order. The artist name is the most significant field according to the natural order defined for CDs (p. 883).

Click here to view code image

```
Optional<CD> minCD = CD.cdList.stream().min(Comparator.naturalOrder());
minCD.ifPresent(out::println);      // <Funkies, "Lambda Dancing", 10, 2018, P
out.println(minCD.map(CD::artist).orElse("No min CD."));    // Funkies

Optional<CD> maxCD = CD.cdList.stream().max(Comparator.naturalOrder());
maxCD.ifPresent(out::println);      // <Jaav, "Java Jive", 8, 2017, POP>
out.println(maxCD.map(CD::artist).orElse("No max CD."));    // Jaav
```

In the code below, the `max()` method is applied to an `IntStream` to find the largest number between 1 and 100 that is divisible by 7.

Click here to view code image

```
IntStream iStream = IntStream.rangeClosed(1, 100);
OptionalInt maxNum = iStream.filter(n -> n % 7 == 0).max(); // 98
```

If one is only interested in the minimum and maximum elements in a collection, the overloaded methods `min()` and `max()` of the `java.util.Collections` class can be more convenient to use.

Implementing Functional Reduction: The `reduce()` Method

A *functional reduction* combines all elements in a stream to produce a *single immutable value* as its result. The reduction process employs an *accumulator* that repeatedly computes a new partial result based on the current partial result and the current element in the stream. The stream thus gets shorter by one element. When all elements have been combined, the last partial result that was computed by the accumulator is returned as the final result of the reduction process.

The following terminal operations are special cases of functional reduction:

- `count()`, p. 953.
- `min()`, p. 954.
- `max()`, p. 954.
- `average()`, p. 1000.

- `sum()`, p. 1001.

The overloaded `reduce()` method can be used to implement new forms of functional reduction.

Click here to view code image

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

This terminal operation returns an `Optional` with the *cumulative* result of applying the `accumulator` on the elements of this stream: $e_1 \oplus e_2 \oplus e_3$ ..., where each e_i is an element of this stream and ⊕ is the `accumulator`. If the stream is empty, an empty `Optional` is returned.

The `accumulator` must be *associative*—that is, the result of evaluating an expression is the same, regardless of how the operands are grouped to evaluate the expression. For example, the grouping in the expression below allows the subexpressions to be evaluated in parallel and their results combined by the `accumulator`:

Click here to view code image

$$e_i \oplus e_j \oplus e_k \oplus e_l == (e_i \oplus e_j) \oplus (e_k \oplus e_l)$$

where e_i, e_j, e_k, and e_l are operands, and ⊕ is the `accumulator`. For example, numeric addition, min, max, and string concatenation are associative operations, whereas subtraction and division are nonassociative.

The `accumulator` must also be a *non-interfering* and *stateless* function (p. 909).

Note that the method reduces a `Stream` of type `T` to a result that is an `Optional` of type `T`.

A counterpart to the single-argument `reduce()` method is also provided for the numeric streams.

Click here to view code image

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

This terminal operation returns the *cumulative* result of applying the `accumulator` on the elements of this stream: `identity` $\oplus e_1 \oplus e_2 \oplus e_3$ ..., where e_i is an element of this stream, and ⊕ is the `accumulator`. The `identity` value is the initial value to accumulate. If the stream is empty, the `identity` value is returned.

The `identity` value must be an *identity* for the `accumulator`—for all e_i, `identity` ⊕ e_i `==` e_i. The `accumulator` must be *associative*.

The `accumulator` must also be a *non-interfering* and *stateless* function (p. 909).

Note that the method reduces a `Stream` of type `T` to a result of type `T`.

A counterpart to the two-argument `reduce()` method is also provided for the numeric streams.

Click here to view code image

```
<U> U reduce(
        U identity,
        BiFunction<U,? super T,U> accumulator,
        BinaryOperator<U> combiner)
```

This terminal operation returns the *cumulative* result of applying the `accumulator` on the elements of this stream, using the `identity` value of type `U` as the initial value to accumulate. If the stream is empty, the `identity` value is returned.

The `identity` value must be an *identity* for the `combiner` function. The `accumulator` and the `combiner` function must also satisfy the following relationship for all `u` and `t` of type `U` and `T`, respectively:

```
  u © (identity ⊕ t) == u ⊕ t
```

where © and ⊕ are the `accumulator` and `combiner` functions, respectively.

The `combiner` function combines two values during stream processing. It may not be executed for a sequential stream, but for a parallel stream, it will combine cumulative results of segments that are processed concurrently.

Both the `accumulator` and the `combiner` must also be *non-interfering* and *stateless* functions (p. 909).

Note that the `accumulator` has the function type `(U, T) -> U`, and the `combiner` function has the function type `(U, U) -> U`, where the type parameters `U` and `T` are always the types of the partial result and the stream element, respectively. This method reduces a `Stream` of type `T` to a result of type `U`.

There is *no* counterpart to the three-argument `reduce()` method for the numeric streams.

The idiom of using a loop for calculating the sum of a finite number of values is something that is ingrained into all aspiring programmers. A loop-based solution to calculate the total number of tracks on CDs in a list is shown below, where the variable `sum` will hold the result after the execution of the `for(:)` loop:

Click here to view code image

```
int sum = 0;                               // (1) Initialize the partial resul
for (CD cd : CD.cdList) {                  // (2) Iterate over the list.
  int numOfTracks = cd.noOfTracks();       // (3) Get the current value.
  sum = sum + numOfTracks;                 // (4) Calculate new partial result
}
```

Apart from the `for(:)` loop at (2) to iterate over all elements of the list and read the number of tracks in each CD at (3), the two necessary steps are:

- Initialization of the variable `sum` at (1)
- The accumulative operation at (4) that is applied repeatedly to compute a new partial result in the variable `sum`, based on its previous value and the number of tracks in the current CD

The loop-based solution above can be translated to a stream-based solution, as shown in Figure 16.11. All the code snippets can be found in Example 16.11.

Figure 16.11 *Reducing with an Initial Value*

In Figure 16.11, the stream created at (6) internalizes the iteration over the elements. The `mapToInt()` intermediate operation maps each CD to its number of tracks at (7)—the `Stream<CD>` is mapped to an `IntStream`. The `reduce()` terminal operation with two arguments computes and returns the total number of tracks:

- Its first argument at (8) is the *identity* element that provides the initial value for the operation and is also the default value to return if the stream is empty. In this case, this value is `0`.
- Its second argument at (9) is the *accumulator* that is implemented as a lambda expression. It repeatedly computes a new partial sum based on the previous partial sum and the number of tracks in the current CD, as evident from

Figure 16.11. In this case, the accumulator is an `IntBinaryOperator`

whose functional type is `(int, int) -> int`. Note that the parameters of the lambda expression represent the partial sum and the current number of tracks, respectively.

The stream pipeline in Figure 16.11 is an example of a *map-reduce transformation* on a sequential stream, as it maps the stream elements first and then reduces them. Typically, a filter operation is also performed before the map-reduce transformation.

Each of the following calls can replace the `reduce()` method call in Figure 16.11, as they are all equivalent:

Click here to view code image

```
reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks))
reduce(0, Integer::sum)                // Method reference
sum()                                  // Special functional reduction, p. 1001
```

In Example 16.11, the stream pipeline at (10) prints the actions taken by the accumulator which is now augmented with print statements. The output at (3) shows that the accumulator actions correspond to those in Figure 16.11.

The single-argument `reduce()` method only accepts an accumulator. As no explicit default or initial value can be specified, this method returns an `Optional`. If the stream is not empty, it uses the first element as the initial value; otherwise, it returns an empty `Optional`. In Example 16.11, the stream pipeline at (13) uses the single-argument `reduce()` method to compute the total number of tracks on CDs. The return value is an `OptionalInt` that can be queried to extract the encapsulated `int` value.

Click here to view code image

```
OptionalInt optSumTracks0 = CD.cdList                        // (13)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce(Integer::sum);                                   // (14)
out.println("Total number of tracks: " + optSumTracks0.orElse(0));  // 42
```

We can again augment the accumulator with print statements as shown at (16) in Example 16.11. The output at (5) shows that the number of tracks from the first CD was used as the initial value before the accumulator is applied repeatedly to the rest of the values.

Example 16.11 *Implementing Functional Reductions*

Click here to view code image

```
import static java.lang.System.out;

import java.util.Comparator;
import java.util.Optional;
import java.util.OptionalInt;
import java.util.function.BinaryOperator;

public final class FunctionalReductions {
  public static void main(String[] args) {

// Two-argument reduce() method:
  {
    out.println("(1) Find total number of tracks (loop-based version):");
    int sum = 0;                               // (1) Initialize the partial result.
    for (CD cd : CD.cdList) {                  // (2) Iterate over the list.
      int numOfTracks = cd.noOfTracks();       // (3) Get the next value.
      sum = sum + numOfTracks;                 // (4) Calculate new partial result.
    }
    out.println("Total number of tracks: " + sum);
  }

    out.println("(2) Find total number of tracks (stream-based version):");
    int totNumOfTracks = CD.cdList                                // (5)
        .stream()                                                 // (6)
        .mapToInt(CD::noOfTracks)                                 // (7)
        .reduce(0,                                                // (8)
                (sum, numOfTracks) -> sum + numOfTracks);         // (9)
    //  .reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks));
    //  .reduce(0, Integer::sum);
    //  .sum();
    out.println("Total number of tracks: " + totNumOfTracks);
    out.println();

    out.println("(3) Find total number of tracks (accumulator logging): ");
    int totNumOfTracks1 = CD.cdList                               // (10)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce(0,                                                // (11)
            (sum, noOfTracks) -> {                                // (12)
                int newSum = sum + noOfTracks;
                out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
                           sum, noOfTracks, newSum);
                return newSum;
            }
         );
    out.println("Total number of tracks: " + totNumOfTracks1);
    out.println();

// One-argument reduce() method:

    out.println("(4) Find total number of tracks (stream-based version):");
    OptionalInt optSumTracks0 = CD.cdList                         // (13)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce(Integer::sum);                                    // (14)
    out.println("Total number of tracks: " + optSumTracks0.orElse(0));
    out.println();

    out.println("(5) Find total number of tracks (accumulator logging): ");
    OptionalInt optSumTracks1 = CD.cdList                         // (15)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce((sum, noOfTracks) -> {                            // (16)
           int newSum = sum + noOfTracks;
           out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
                       sum, noOfTracks, newSum);
           return newSum;
         });
    out.println("Total number of tracks: " + optSumTracks1.orElse(0));
    out.println();

// Three-argument reduce() method:

    out.println("(6) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks5 = CD.cdList                         // (17)
    //  .stream()                                          // (18a)
        .parallelStream()                                  // (18b)
        .reduce(Integer.valueOf(0),                        // (19) Initial value
                (sum, cd) -> sum + cd.noOfTracks(),        // (20) Accumulator
                (sum1, sum2) -> sum1 + sum2);              // (21) Combiner
    out.println("Total number of tracks: " + sumTracks5);
    out.println();

    out.println("(7) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks6 = CD.cdList                         // (22)
//      .stream()                                          // (23a)
        .parallelStream()                                  // (23b)
        .reduce(0,
               (sum, cd) -> {                              // (24) Accumulator
                 Integer noOfTracks = cd.noOfTracks();
                 Integer newSum = sum + noOfTracks;
                 out.printf("Accumulator: sum=%2d, noOfTracks=%2d, "
                            + "newSum=%2d%n", sum, noOfTracks, newSum);
                 return newSum;
               },
               (sum1, sum2) -> {                           // (25) Combiner
                 Integer newSum = sum1 + sum2;
                 out.printf("Combiner: sum1=%2d, sum2=%2d, newSum=%2d%n",
                            sum1, sum2, newSum);
                 return newSum;
               }
         );
    out.println("Total number of tracks: " + sumTracks6);
    out.println();

    // Compare by CD title.
    Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);    // (26)
    BinaryOperator<CD> maxByTitle =
        (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

    // Query: Find maximum Jazz CD by title:
    Optional<CD> optMaxJazzCD = CD.cdList                  // (28)
        .stream()
        .filter(CD::isJazz)
        .reduce(BinaryOperator.maxBy(cmpByTitle));         // (29a)
    //  .reduce(maxByTitle);                               // (29b)
    //  .max(cmpByTitle);                                  // (29c)
    optMaxJazzCD.map(CD::title).ifPresent(out::println);// Keep on Erasing
  }
}
```

Possible output from the program:

Click here to view code image

```
(1) Find total number of tracks (loop-based version):
```

```
Total number of tracks: 42
(2) Find total number of tracks (stream-based version):
Total number of tracks: 42

(3) Find total number of tracks (accumulator logging):
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(4) Find total number of tracks (stream-based version):
Total number of tracks: 42

(5) Find total number of tracks (accumulator logging):
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(6) Find total number of tracks (accumulator + combiner):
Total number of tracks: 42

(7) Find total number of tracks (accumulator + combiner):
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2=10, newSum=18
Combiner: sum1=10, sum2=18, newSum=28
Accumulator: sum= 0, noOfTracks= 6, newSum= 6
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2= 6, newSum=14
Combiner: sum1=14, sum2=28, newSum=42
Total number of tracks: 42

Keep on Erasing
```

The single-argument and two-argument `reduce()` methods accept a *binary operator* as the accumulator whose arguments and result are of the *same* type. The three-argument `reduce()` method is more flexible and can only be applied to objects. The stream pipeline below computes the total number of tracks on CDs using the three-argument `reduce()` method.

Click here to view code image

```
Integer sumTracks5 = CD.cdList                               // (17)
    .stream()                                                // (18a)
//  .parallelStream()                                        // (18b)
    .reduce(Integer.valueOf(0),                              // (19) Initial value
            (sum, cd) -> sum + cd.noOfTracks(),              // (20) Accumulator
            (sum1, sum2) -> sum1 + sum2);                    // (21) Combiner
```

The `reduce()` method above accepts the following arguments:

- An *identity* value: Its type is `U`. In this case, it is an `Integer` that wraps the value `0`. As before, it is used as the initial value. The type of the value returned by the `reduce()` method is also `U`.
- An *accumulator*: It is a `BiFunction<U,T,U>`; that is, it is a

binary function that accepts an object of type `U` and an object of type `T` and produces a result of type `U`. In this case, type `U` is `Integer` and type `T` is `CD`. The lambda expression implementing the accumulator first reads the number of tracks from the current CD before the addition operator is applied. Thus the accumulator will calculate the sum of `Integer`s which are, of course, unboxed and boxed to do the calculation. As we have seen earlier, the accumulator is repeatedly applied to sum the tracks on the CDs. Only this time, the mapping of a `CD` to an `Integer` is done when the accumulator is evaluated.

- A *combiner*: It is a `BinaryOperator<U>`; that is, it is a binary operator whose arguments and result are of the same type `U`. In this case, type `U` is `Integer`. Thus the combiner will calculate the sum of `Integer`s which are unboxed and boxed to do the calculation.

  In the code above, the combiner is not executed if the `reduce()` method is applied to a *sequential* stream. However, there is no guarantee that this is always the case for a sequential stream. If we uncomment (18b) and remove (18a), the combiner will be executed on the *parallel* stream.

That the combiner in the three-argument `reduce()` method is executed for a parallel stream is illustrated by the stream pipeline at (22) in Example 16.11, that has been augmented with print statements. There is no output from the combiner when the stream is sequential. The output at (7) in Example 16.11 shows that the combiner accumulates the partial sums created by the accumulator when the stream is parallel.

Parallel Functional Reduction

Parallel execution is illustrated in Figure 16.12. Multiple instances of the stream pipeline are executed in parallel, where each pipeline instance processes a segment of the stream. In this case, only one CD is allocated to each pipeline instance. Each pipeline instance thus produces its partial sum, and the combiner is applied in parallel on the partial sums to combine them into a final result. No additional synchronization is required to run the `reduce()` operation in parallel.

Figure 16.12 also illustrates why the initial value must be an identity value. Say we had specified the initial value to be 3. Then the value 3 would be added multiple times to the sum during parallel execution. We also see why both the accumulator and the combiner are associative, as this allows for any two values to be combined in any order.

Figure 16.12 *Parallel Functional Reduction*

When the single-argument and two-argument `reduce()` methods are applied to a parallel stream, the accumulator also acts as the combiner. The three-argument `reduce()` method can usually be replaced with a *map-reduce* transformation, making the combiner redundant.

We conclude the discussion on implementing functional reductions by implementing the `max()` method that finds the maximum element in a stream according to a given comparator. A comparator that compares by the CD title is defined at (26). A binary operator that finds the maximum of two CDs when compared by title is defined at (27). It uses the comparator defined at (26). The stream pipeline at (28) finds the maximum of all jazz music CDs by title. The method calls at (29a), (29b), and (29c) are equivalent.

Click here to view code image

```
Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);    // (26)
BinaryOperator<CD> maxByTitle =
    (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

Optional<CD> optMaxJazzCD = CD.cdList                   // (28)
    .stream()
    .filter(CD::isJazz)
    .reduce(BinaryOperator.maxBy(cmpByTitle));          // (29a)
//  .reduce(maxByTitle);                                // (29b)
//  .max(cmpByTitle);                                   // (29c)
optMaxJazzCD.map(CD::title).ifPresent(out::println);    // Keep on Erasing
```

The accumulator at (29a), returned by the `BinaryOperator.maxBy()` method, will compare the previous maximum CD and the current CD by title to compute a new maximum jazz music CD. The accumulator used at (29b) is implemented at (27). It also does the same comparison as the accumulator at (29a). At (29c), the `max()` method also does the same thing, based on the comparator at (26). Note that the return value is an `Optional<CD>`, as the stream might be empty. The `Optional<CD>` is mapped to an `Optional<String>`. If it is not empty, its value—that is, the CD title— is printed.

The `reduce()` method does not terminate if applied to an infinite stream, as the method will never finish processing all stream elements.

Implementing Mutable Reduction: The `collect()` Method

The `collect(Collector)` method accepts a *collector* that encapsulates the functions required to perform a mutable reduction. We discuss predefined collectors implemented by the `java.util.stream.Collectors` class in a later section (p. 978). The code below uses the collector returned by the `Collectors.toList()` method that accumulates the result in a list (p. 980).

Click here to view code image

```
List<String> titles = CD.cdList.stream()
                            .map(CD::title).collect(Collectors.toList());
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

The `collect(supplier, accumulator, combiner)` generic method provides the general setup for implementing mutable reduction on stream elements using different kinds of *mutable containers*—for example, a list, a map, or a `StringBuilder`. It uses one or more mutable containers to accumulate partial results that are combined into a single mutable container that is returned as the result of the reduction operation.

Click here to view code image

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

This terminal operation performs a reduction operation on the elements of this stream using a `Collector` (p. 978).

A `Collector` encapsulates the functions required for performing the reduction.

The result of the reduction is of type `R`, and the type parameter `A` is the intermediate accumulation type of the `Collector`.

Click here to view code image

```
<R> R collect(
          Supplier<R>                supplier,
          BiConsumer<R,? super T>    accumulator,
          BiConsumer<R,R>            combiner)
```

This terminal operation performs a mutable reduction on the elements of this stream. A counterpart to this method is also provided for numeric streams.

The `supplier` creates a new mutable container of type `R`—

> which is typically empty. Elements are incorporated into such a container during the reduction process. For a parallel stream, the supplier can be called multiple times, and the container returned by the supplier must be an *identity container* in the sense that it does not mutate any result container with which it is merged.
>
> The `accumulator` incorporates additional elements into a result container: A stream element of type `T` is incorporated into a mutable container of type `R`.
>
> The `combiner` merges two values that are mutable containers of type `R`. It must be compatible with the `accumulator`. There is no guarantee that the `combiner` is called if the stream is sequential, but definitely comes into play if the stream is parallel.
>
> Both the `accumulator` and the `combiner` must also be *non-interfering* and *stateless* functions (p. 909).
>
> With the above requirements on the argument functions fulfilled, the `collect()` method will produce the same result regardless of whether the stream is sequential or parallel.

We will use Figure 16.13 to illustrate mutable reduction performed on a sequential stream by the three-argument `collect()` method. The figure shows both the code and the execution of a stream pipeline to create a list containing the number of tracks on each CD. The stream of `CD`s is mapped to a stream of `Integer`s at (3), where each `Integer` value is the number of tracks on a `CD`. The `collect()` method at (4) accepts three functions as arguments. They are explicitly defined as lambda expressions to show what the parameters represent and how they are used to perform mutable reduction. Implementation of these functions using method references can be found in Example 16.12.

- *Supplier*: The supplier is a `Supplier<R>` that is used to create new instances of a mutable result container of type `R`. Such a container holds the results computed by the accumulator and the combiner. In Figure 16.13, the supplier at (4) returns an *empty* `ArrayList<Integer>` every time it is called.
- *Accumulator*: The accumulator is a `BiConsumer<R, T>` that is used to accumulate an element of type `T` into a mutable result container of type `R`. In Figure 16.13, type `R` is

`ArrayList<Integer>` and type `T` is `Integer`. The accumulator at (5) mutates a container of type `ArrayList<Integer>` by repeatedly adding a new `Integer` value to it, as illustrated in Figure 16.13b. It is instructive to contrast this accumulator with the accumulator for sequential functional reduction illustrated in Figure 16.11, p. 957.

- *Combiner*: The combiner is a `BiConsumer<R, R>` that merges the contents of the second argument container with the contents of the first argument container, where both containers are of type `R`. As in the case of the `reduce(identity, accumulator, combiner)` method, the combiner is executed when the `collect()` method is called on a parallel stream.

Figure 16.13 *Sequential Mutable Reduction*

Parallel Mutable Reduction

Figure 16.14 shows the stream pipeline from Figure 16.13, where the sequential stream (2a) has been replaced by a parallel stream (2b); in other words, the `collect()` method is called on a parallel stream. One possible parallel execution of the pipeline is also depicted in Figure 16.14b. We see five instances of the pipeline being executed in parallel. The supplier creates five empty `ArrayList`s that are used as partial result containers by the accumulator, and are later merged by the combiner to a final result container. The containers created by the supplier are mutated by the accumulator and the combiner to perform mutable reduction. The partial result containers are also merged in parallel by the combiner. It is instructive to contrast this combiner with the combiner for parallel functional reduction that is illustrated in Figure 16.12, p. 963.

Figure 16.14 *Parallel Mutable Reduction*

In Example 16.12, the stream pipeline at (7) also creates a list containing the number of tracks on each CD, where the stream is parallel, and the lambda expressions implementing the argument functions of the `collect()` method are augmented with print statements so that actions of the functions can be logged. The output from this parallel mutable reduction shows that the combiner is executed multiple times to merge partial result lists. The actions of the argument functions shown in the output are the same as those illustrated in Figure 16.14b. Of course, multiple runs of the pipeline can show different sequences of operations in the output, but the final result in the same. Also note that the elements retain their relative position in the partial result lists as these are combined, preserving the encounter order of the stream.

Although a stream is executed in parallel to perform mutable reduction, the merging of the partial containers by the combiner can impact performance if this is too costly. For example, merging mutable maps can be costly compared to merging mutable lists. This issue is further explored for parallel streams in §16.9, p. 1009.

Example 16.12 *Implementing Mutable Reductions*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Stream;

public final class Collecting {
  public static void main(String[] args) {

    // Query: Create a list with the number of tracks on each CD.
    System.out.println("Sequential Mutable Reduction:");
    List<Integer> tracks = CD.cdList                         // (1)
        .stream()                                            // (2a)
//      .parallelStream()                                    // (2b)
        .map(CD::noOfTracks)                                 // (3)
        .collect(() -> new ArrayList<>(),                    // (4) Supplier
                 (cont, noOfTracks) -> cont.add(noOfTracks), // (5) Accumulat
                 (cont1, cont2) -> cont1.addAll(cont2));     // (6) Combiner
//      .collect(ArrayList::new, ArrayList::add, ArrayList::addAll); // (6a)
//      .toList();
    System.out.println("Number of tracks on each CD (sequential): " + tracks);
    System.out.println();

    System.out.println("Parallel Mutable Reduction:");
    List<Integer> tracks1 = CD.cdList                        // (7)
//      .stream()                                            // (8a)
        .parallelStream()                                    // (8b)
        .map(CD::noOfTracks)                                 // (9)
        .collect(                                            // (10)
            () -> {                                          // (11) Supplier
              System.out.println("Supplier: Creating an ArrayList");
              return new ArrayList<>();
            },
            (cont, noOfTracks) -> {                          // (12) Accumula
              System.out.printf("Accumulator: cont:%s, noOfTracks:%s",
                                 cont, noOfTracks);
              cont.add(noOfTracks);
              System.out.printf(", mutCont:%s%n", cont);
            },
            (cont1, cont2) -> {                              // (13) Combiner
              System.out.printf("Combiner: con1:%s, cont2:%s", cont1, cont2);
              cont1.addAll(cont2);
              System.out.printf(", mutCont:%s%n", cont1);
            });
    System.out.println("Number of tracks on each CD (parallel): " + tracks1);
    System.out.println();

    // Query: Create an ordered set with CD titles, according to natural order
    Set<String> cdTitles = CD.cdList                          // (14)
        .stream()
        .map(CD::title)
        .collect(TreeSet::new, TreeSet::add, TreeSet::addAll);// (15)
    System.out.println("CD titles: " + cdTitles);
    System.out.println();

    // Query: Go bananas.
    StringBuilder goneBananas = Stream                        // (16)
        .iterate("ba", b -> b + "na")                         // (17)
        .limit(5)
        .peek(System.out::println)
        .collect(StringBuilder::new,                          // (18)
                 StringBuilder::append,
                 StringBuilder::append);
    System.out.println("Go bananas: " + goneBananas);
  }
}
```

Possible output from the program:

Click here to view code image

```
Sequential Mutable Reduction:
Number of tracks on each CD (sequential): [8, 6, 10, 8, 10]

Parallel Mutable Reduction:
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:6, mutCont:[6]
Combiner: con1:[8], cont2:[6], mutCont:[8, 6]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Combiner: con1:[10], cont2:[8], mutCont:[10, 8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Combiner: con1:[10, 8], cont2:[10], mutCont:[10, 8, 10]
Combiner: con1:[8, 6], cont2:[10, 8, 10], mutCont:[8, 6, 10, 8, 10]
Number of tracks on each CD (parallel): [8, 6, 10, 8, 10]

CD titles: [Hot Generics, Java Jam, Java Jive, Keep on Erasing, Lambda Dancing

ba
bana
banana
bananana
banananana
Go bananas: babanabananabanananabanananana
```

Example 16.12 also shows how other kinds of containers can be used for mutable reduction. The stream pipeline at (14) performs mutable reduction to create an ordered set with CD titles. The supplier is implemented by the constructor reference `TreeSet::new`. The constructor will create a container of type `TreeSet<String>`

that will maintain the CD titles according to the natural order for `String`s. The accumulator and the combiner are implemented by the method references `TreeSet::add` and `TreeSet::addAll`, respectively. The accumulator will add a title to a container of type `TreeSet<String>` and the combiner will merge the contents of two containers of type `TreeSet<String>`.

In Example 16.12, the mutable reduction performed by the stream pipeline at (16) uses a mutable container of type `StringBuilder`. The output from the `peek()` method shows that the strings produced by the `iterate()` method start with the initial string `"ba"` and are iteratively concatenated with the postfix `"na"`. The `limit()` intermediate operation truncates the infinite stream to five elements. The `collect()` method appends the strings to a `StringBuilder`. The supplier creates an empty `StringBuilder`. The accumulator and the combiner append a `CharSequence` to a `StringBuilder`. In the case of the accumulator, the `CharSequence` is a `String`—that is, a stream element—in the call to the `append()` method. But in the case of the combiner, the `CharSequence` is a `StringBuilder`—that is, a partial result container when the stream is parallel. One might be tempted to use a string instead of a `StringBuilder`, but that would not be a good idea as a string is immutable.

Note that the accumulator and combiner of the `collect()` method do not return a value. The `collect()` method does not terminate if applied to an infinite stream, as the method will never finish processing all the elements in the stream.

Because mutable reduction uses the same mutable result container for accumulating new results by changing the state of the container, it is more efficient than a functional reduction where a new partial result always replaces the previous partial result.

Collecting to an Array

The overloaded method `toArray()` can be used to collect or accumulate into an array. It is a special case of a mutable reduction, and as the name suggests, the mutable container is an array. The numeric stream interfaces also provide a counterpart to the `toArray()` method that returns an array of a numeric type.

> `Object[] toArray()`
>
> This terminal operation returns an array containing the elements of this stream. Note that the array returned is of type

> `Object[]`.
>
> Click here to view code image
>
> ```
> <A> A[] toArray(IntFunction<A[]> generator)
> ```
>
> This terminal operation returns an array containing the elements of this stream. The provided `generator` function is used to allocate the desired array. The type parameter `A` is the element type of the array that is returned. The size of the array (which is equal to the length of the stream) is passed to the `generator` function as an argument.

The zero-argument method `toArray()` returns an array of objects, `Object[]`, as generic arrays cannot be created at runtime. The method needs to store all the elements before creating an array of the appropriate length. The query at (1) finds the titles of the CDs, and the `toArray()` method collects them into an array of objects, `Object[]`.

Click here to view code image

```
Object[] objArray = CD.cdList.stream().map(CD::title)
                                        .toArray();                      // (1)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

The `toArray(IntFunction<A>)` method accepts a generator function that creates an array of type `A`, (`A[]`), whose length is passed as a parameter by the method to the generator function. The array length is determined from the number of elements in the stream. The query at (2) also finds the CD titles, but the `toArray()` method collects them into an array of strings, `String[]`. The method reference defining the generator function is equivalent to the lambda expression `(len -> new String[len])`.

Click here to view code image

```
String[] cdTitles = CD.cdList.stream().map(CD::title)
                                        .toArray(String[]::new);          // (2)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

Examples of numeric streams whose elements are collected into an array are shown at (3) and (4). The `limit()` intermediate operation at (3) converts the infinite stream into a finite one whose elements are collected into an `int` array.

Click here to view code image

```
int[] intArray1 = IntStream.iterate(1, i -> i + 1).limit(5).toArray();// (3)
// [1, 2, 3, 4, 5]
```

```
int[] intArray2 = IntStream.range(-5, 5).toArray();                         // (4)
// [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Not surprisingly, when applied to infinite streams the operation results in a fatal `OutOfMemoryError`, as the method cannot determine the length of the array and keeps storing the stream elements, eventually running out of memory.

Click here to view code image

```
int[] intArray3 = IntStream.iterate(1, i -> i + 1)                          // (5)
                           .toArray();                             // OutOfMemoryEr
```

If the sole purpose of using the `toArray()` operation in a pipeline is to convert the data source collection to an array, it is far better to use the overloaded `Collection.toArray()` methods. For one thing, the size of the array is easily determined from the size of the collection.

Click here to view code image

```
CD[] cdArray1 = CD.cdList.stream().toArray(CD[]::new);        // (6) Preferred.
CD[] cdArray2 = CD.cdList.toArray(new CD[CD.cdList.size()]); // (7) Not effici
```

Like any other mutable reduction operation, the `toArray()` method does not terminate when applied to an infinite stream, unless it is converted into a finite stream as at (3) above.

Collecting to a List

The method `Stream.toList()` implements a terminal operation that can be used to collect or accumulate the result of processing a stream into a list. Compared to the `toArray()` instance method, the `toList()` method is a `default` method in the `Stream` interface. The default implementation returns an *unmodifiable* list; that is, elements cannot be added, removed, or sorted. This unmodifiable list is created from the array into which the elements are accumulated first.

If the requirement is an unmodifiable list that allows `null` elements, the `Stream.to-List()` is the clear and concise choice. Many examples of stream pipelines encountered so far in this chapter use the `toList()` terminal operation.

Click here to view code image

```
List<String> titles = CD.cdList.stream().map(CD::title).toList();
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles");          // UnsupportedOperationException!
```

Like any other mutable reduction operation, the `toList()` method does not terminate when applied to an infinite stream, unless the stream is converted into a finite stream.

> `default List<T> toList()`
>
> Accumulates the elements of this stream into a `List`, respecting any encounter order the stream may have. The returned `List` is *unmodifiable* (§12.2, p. 649), and calls to any mutator method will always result in an `UnsupportedOperation-Exception`. The unmodifiable list returned allows `null` values.
>
> See also the `toList()` method in the `Collectors` class (p. 980).
>
> The `Collectors.toCollection(Supplier)` method is recommended for greater control.

Functional Reductions Exclusive to Numeric Streams

In addition to the counterparts of the methods in the `Stream<T>` interface, the following functional reductions are exclusive to the numeric stream interfaces `IntStream`, `LongStream`, and `DoubleStream`. These reduction operations are designed to calculate various statistics on numeric streams.

In the methods below, *`NumType`* is `Int`, `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`. These statistical operations do not terminate when applied to an infinite stream:

> *`numtype`* `sum()`
>
> This terminal operation returns the sum of elements in this stream. It returns zero if the stream is empty.
>
> `OptionalDouble average()`
>
> This terminal operation returns an `OptionalDouble` that encapsulates the arithmetic mean of elements of this stream, or an empty `Optional` if this stream is empty.
>
> Click here to view code image
>
> *`NumType`*`SummaryStatistics summaryStatistics()`
>
> This terminal operation returns a *`NumType`*`SummaryStatistics` describing various summary data about the elements of this

stream.

Summation

The `sum()` terminal operation is a special case of a functional reduction that calculates the sum of numeric values in a stream. The stream pipeline below calculates the total number of tracks on the CDs in a list. Note that the stream of `CD` is mapped to an `int` stream whose elements represent the number of tracks on a CD. The `int` values are cumulatively added to compute the total number of tracks.

Click here to view code image

```
int totNumOfTracks = CD.cdList
    .stream()                                       // Stream<CD>
    .mapToInt(CD::noOfTracks)                       // IntStream
    .sum();                                         // 42
```

The query below sums all even numbers between 1 and 100.

Click here to view code image

```
int sumEven = IntStream
    .rangeClosed(1, 100)
    .filter(i -> i % 2 == 0)
    .sum();                                         // 2550
```

The `count()` operation is equivalent to mapping each stream element to the value 1 and adding the 1s:

Click here to view code image

```
int numOfCDs = CD.cdList
    .stream()
    .mapToInt(cd -> 1)                              // CD => 1
    .sum();                                         // 5
```

For an empty stream, the sum is always zero.

Click here to view code image

```
double total = DoubleStream.empty().sum();          // 0.0
```

Averaging

Another common statistics to calculate is the average of values, defined as the sum of values divided by the number of values. A loop-based solution to calculate the average would explicitly sum the values, count the number of values, and do the calculation. In a stream-based solution, the `average()` terminal operation can be used to calculate this value. The stream pipeline below computes the average number of tracks on a CD. The CD stream is mapped to an `int` stream whose values are the number of tracks on a CD. The `average()` terminal operation adds the number of tracks and counts the values, returning the average as a `double` value encapsulated in an `OptionalDouble`.

Click here to view code image

```
OptionalDouble optAverage = CD.cdList
    .stream()
    .mapToInt(CD::noOfTracks)
    .average();
System.out.println(optAverage.orElse(0.0));        // 8.4
```

The reason for using an `Optional` is that the average is not defined if there are no values. The absence of a value in the `OptionalDouble` returned by the method means that the stream was empty.

Summarizing

The result of a functional reduction is a single value. This means that for calculating different results—for example, count, sum, average, min, and max—requires separate reduction operations on a stream.

The method `summaryStatistics()` does several common reductions on a stream in a single operation and returns the results in an object of type *`NumType`*`SummaryStatistics`, where *`NumType`* is `Int`, `Long`, or `Double`. An object of this class encapsulates the count, sum, average, min, and max values of a stream.

The classes `IntSummaryStatistics`, `LongSummaryStatistics`, and `DoubleSummaryStatistics` in the `java.util` package define the following constructor and methods, where *`NumType`* is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`:

> *`NumType`*`SummaryStatistics()`
>
> Creates an empty instance with zero count, zero sum, a min value as *Num-Type*.MAX_VALUE, a max value as *`NumType`*`.MIN_VALUE`, and an average value of zero.
>
> `double getAverage()`
>
> Returns the arithmetic mean of values recorded, or zero if no

> values have been recorded.
>
> `long getCount()`
>
> Returns the count of values recorded.
>
> `numtype getMax()`
>
> Returns the maximum value recorded, or `NumType.MIN_VALUE` if no values have been recorded.
>
> `numtype getMin()`
>
> Returns the minimum value recorded, or `NumType.MAX_VALUE` if no values have been recorded.
>
> `numtype getSum()`
>
> Returns the sum of values recorded, or zero if no values have been recorded. The method in the `IntSummaryStatistics` and `LongSummaryStatistics` classes returns a `long` value. The method in the `DoubleSummaryStatistics` class returns a `double` value.
>
> `void accept(numtype value)`
>
> Records a new value into the summary information, and updates the various statistics. The method in the `LongSummaryStatistics` class is overloaded and can accept an `int` value as well.
>
> Click here to view code image
>
> `void combine(NumTypeSummaryStatistics other)`
>
> Combines the state of another `NumTypeSummaryStatistics` into this one.

The `summaryStatistics()` method is used to calculate various statistics for the number of tracks on two CDs processed by the stream pipeline below. Various get methods are called on the `IntSummaryStatistics` object returned by the `summaryStatistics()` method, and the statistics are printed.

Click here to view code image

```
IntSummaryStatistics stats1 = List.of(CD.cd0, CD.cd1)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println("Count="   + stats1.getCount());         // Count=2
System.out.println("Sum="     + stats1.getSum());           // Sum=14
System.out.println("Min="     + stats1.getMin());           // Min=6
System.out.println("Max="     + stats1.getMax());           // Max=8
System.out.println("Average=" + stats1.getAverage());      // Average=7.0
```

The default format of the statistics printed by the `toString()` method of the `IntSummaryStatistics` class is shown below:

Click here to view code image

```
System.out.println(stats1);
//IntSummaryStatistics{count=2, sum=14, min=6, average=7.000000, max=8}
```

Below, the `accept()` method records the value `10` (the number of tracks on `CD.cd2`) into the summary information referenced by `stats1`. The resulting statistics show the new count is `3` (=2 +1), the new sum is `24` (=14+10), and the new average is `8.0` (=24.0/3.0). However, the min value was not affected but the max value has changed to 10.

Click here to view code image

```
stats1.accept(CD.cd2.noOfTracks());     // Add the value 10.
System.out.println(stats1);
//IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

The code below creates another `IntSummaryStatistics` object that summarizes the statistics from two other CDs.

Click here to view code image

```
IntSummaryStatistics stats2 = List.of(CD.cd3, CD.cd4)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println(stats2);
//IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10}
```

The `combine()` method incorporates the state of one `IntSummaryStatistics` object into another `IntSummaryStatistics` object. In the code below, the state of the `IntSummary-Statistics` object referenced by `stats2` is combined with the state of the `IntSummary-Statistics` object referenced by `stats1`. The resulting summary information is printed, showing that the new count is `5` (=3 +2), the new sum is `42` (=24+18), and the new average is `8.4` (=42.0/5.0). However, the min and max values were not affected.

Click here to view code image

```
stats1.combine(stats2);                 // Combine stats2 with stats1.
System.out.println(stats1);
//IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

Calling the `summaryStatistics()` method on an empty stream returns an instance of the `IntSummaryStatistics` class with a zero value set for all statistics, except for the min and max values, which are set to `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, respectively. The `IntSummaryStatistics` class provides a zero-argument constructor that also returns an empty instance.

Click here to view code image

```
IntSummaryStatistics emptyStats = IntStream.empty().summaryStatistics();
System.out.println(emptyStats);
//IntSummaryStatistics{count=0, sum=0, min=2147483647, average=0.000000,
//max=-2147483648}
```

The summary statistics classes are not exclusive for use with streams, as they provide a constructor and appropriate methods to incorporate numeric values in order to calculate common statistics, as we have seen here. We will return to calculating statistics when we discuss built-in collectors (p. 978).

Summary of Terminal Stream Operations

The terminal operations of the `Stream<T>` class are summarized in Table 16.5. The type parameter declarations have been simplified, where any bound `<? super T>` or `<? extends T>` has been replaced by `<T>`, without impacting the intent of a method. A reference is provided to each method in the first column.

The last column in Table 16.5 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior of an operation. For example, the method `allMatch`() returns a `boolean` value to indicate whether all elements of a stream satisfy a given predicate. This predicate is implemented as a functional interface `Predicate<T>` that is applied to each element in the stream.

The interfaces `IntStream`, `LongStream`, and `DoubleStream` define analogous methods to those shown for the `Stream<T>` interface in Table 16.5. Methods that are only defined by the numeric stream interfaces are shown in Table 16.6.

Table 16.5 *Terminal Stream Operations*

Method name (ref.)	Any type parameter	Return type	Functional interface parameters	Function type of parameters
`forEach` (p. 948)		`void`	`(Consumer<T> action)`	`T -> void`
`forEachOrdered` (p. 948)		`void`	`(Consumer<T> action)`	`T -> void`
`allMatch` (p. 949)		`boolean`	`(Predicate<T> predicate)`	`T -> boolean`

`anyMatch` (p. 949)	`boolean`	`(Predicate<T> predicate)`
`noneMatch` (p. 949)	`boolean`	`(Predicate<T> predicate)`
`findAny` (p. 952)	`Optional<T>`	`()`
`findFirst` (p. 952)	`Optional<T>`	`()`
`count` (p. 953)	`long`	`()`
`max` (p. 954)	`Optional<T>`	`(Comparator<T> cmp)`
`min` (p. 954)	`Optional<T>`	`(Comparator<T> cmp)`
`reduce` (p. 955)	`Optional<T>`	`(BinaryOperator<T> accumulator)`
`reduce` (p. 955)	`T`	`(T identity,` `BinaryOperator<T> accumulator)`

`reduce` (p. 955)	`U`	`(U identity,`
		`BiFunction<U,T,U> accumulator,`
		`BinaryOperator<U> combiner)`

`collect` (p. 964)	`R`	`(Collector<T,A,R> collector)` Parameter is not a functional interface.

`collect` (p. 964)	`R`	`(Supplier<R> supplier,`
		`BiConsumer<R,T> accumulator,`
		`BiConsumer<R,R> combiner)`

`toArray` (p. 971)	`Object[]`	`()`
`toArray` (p. 971)	`A[]`	`(IntFunction<A[]> generator)`
`toList` (p. 972)	`List<T>`	`()`

Table 16.6 *Additional Terminal Operations in the Numeric Stream Interfaces*

Method name (ref.)	Return type
`average` (p. 948)	`Optional`*`NumType`*, where *`NumType`* is `Int`, `Long`, or `Double`
`sum` (p. 949)	*`numtype`*, where *`numtype`* is `int`, `long`, or `double`
`summaryStatistics` (p. 974)	*`NumType`*`SummaryStatistics`, where *`NumType`* is `Int`, `Long`, or `Double`

16.8 Collectors

A *collector* encapsulates the functions required for performing reduction: the supplier, the accumulator, the combiner, and the finisher. It can provide these functions since it implements the `Collector` interface (in the `java.util.stream` package) that defines the methods to create these functions. It is passed as an argument to the `collect(Collector)` method in order to perform a reduction operation. In contrast, the `collect(Supplier, BiConsumer, BiConsumer)` method requires the functions supplier, accumulator, and combiner, respectively, to be passed as arguments in the method call.

Details of implementing a collector are not necessary for our purposes, as we will exclusively use the extensive set of predefined collectors provided by the static factory methods of the `Collectors` class in the `java.util.stream` package (Table 16.7, p. 1005). In most cases, it should be possible to find a predefined collector for the task at hand. The collectors use various kinds of containers for performing reduction— for example, accumulating to a map, or finding the minimum or maximum element. For example, the `Collectors.toList()` factory method creates a collector that performs mutable reduction using a list as a mutable container. It can be passed to the `collect(Collector)` terminal operation of a stream.

It is a common practice to import the static factory methods of the `Collectors` class in the code so that the methods can be called by their simple names.

Click here to view code image

```
import static java.util.stream.Collectors.*;
```

However, the practice adopted in this chapter is to assume that only the `Collectors` class is imported, enforcing the connection between the static methods and the class to be done explicitly in the code. Of course, static import of factory methods can be used once familiarity with the collectors is established.

Click here to view code image

```
import java.util.stream.Collectors;
```

The three-argument `collect()` method is primarily used to implement mutable reduction, whereas the `Collectors` class provides collectors for both functional and mutable reduction that can be either used in a stand-alone capacity or composed with other collectors.

One group of collectors is designed to collect to a *predetermined container*, which is evident from the name of the static factory method that creates it: `toCollection`, `toList`, `toSet`, and `toMap` (p. 979). The overloaded `toCollection()` and `toMap()` methods allow a specific implementation of a collection and a map to be used, respectively—for example, a `TreeSet` for a collection and a `TreeMap` for a map. In addition, there is the `joining()` method that creates a collector for concatenating the input elements to a `String`—however, internally it uses a mutable `StringBuilder` (p. 984).

Collectors can be composed with other collectors; that is, the partial results from one collector can be additionally processed by another collector (called the *downstream collector*) to produce the final result. Many collectors that can be used as a downstream collector perform functional reduction such as counting values, finding the minimum and maximum values, summing values, averaging values, and summarizing common statistics for values (p. 998).

Composition of collectors is utilized to perform *multilevel grouping* and *partitioning* on stream elements (p. 985). The `groupingBy()` and `partitionBy()` methods return composed collectors to create *classification maps*. In such a map, the keys are determined by a *classifier function*, and the values are the result of a downstream collector, called the *classification mapping*. For example, the CDs in a stream could be classified into a map where the key represents the number of tracks on a CD and the associated value of a key can be a list of CDs with the same number of tracks. The list of CDs with the same number of tracks is the result of an appropriate downstream collector.

Collecting to a `Collection`

The method `toCollection(Supplier)` creates a collector that uses a mutable container of a specific `Collection` type to perform mutable reduction. A supplier to create the mutable container is specified as an argument to the method.

The following stream pipeline creates an `ArrayList<String>` instance with the titles of all CDs in the stream. The constructor reference `ArrayList::new` returns an empty `ArrayList<String>` instance, where the element type `String` is inferred from the context.

Click here to view code image

```
ArrayList<String> cdTitles1 = CD.cdList.stream() // Stream<CD>
    .map(CD::title)                              // Stream<String>
    .collect(Collectors.toCollection(ArrayList::new));
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

> Click here to view code image
>
> ```
> static <T,C extends Collection<T>> Collector<T,?,C>
>        toCollection(Supplier<C> collectionFactory)
> ```
>
> Returns a `Collector` that accumulates the input elements of type `T` into a new `Collection` of type `C`, in encounter order. A new empty `Collection` of type `C` is created by the `collectionFactory` supplier, thus the collection created can be of a specific `Collection` type.
>
> Click here to view code image

```
static <T> Collector<T,?,List<T>> toList()
static <T> Collector<T,?,List<T>> toUnmodifiableList()
```

Return a `Collector` that accumulates the input elements of type `T` into a new `List` or an unmodifiable `List` of type `T`, respectively, in encounter order.

The `toList()` method gives no guarantees of any kind for the returned list.

The unmodifiable list returned does not allow `null` values.

See also the `Stream.toList()` terminal operation (p. 972).

Click here to view code image

```
static <T> Collector<T,?,Set<T>> toSet()
static <T> Collector<T,?,Set<T>> toUnmodifiableSet()
```

Return an unordered `Collector` that accumulates the input elements of type `T` into a new `Set` or an unmodifiable `Set` of type `T`, respectively.

Collecting to a `List`

The method `toList()` creates a collector that uses a mutable container of type `List` to perform mutable reduction. This collector guarantees to preserve the encounter order of the input stream, if it has one. For more control over the type of the list, the `toCollection()` method can be used. This collector can be used as a *downstream collector*.

The following stream pipeline creates a list with the titles of all CDs in the stream using a collector returned by the `Collectors.toList()` method. Although the returned list is modified, this is implementation dependent and should not be relied upon.

Click here to view code image

```
List<String> cdTitles3 = CD.cdList.stream()       // Stream<CD>
    .map(CD::title)                               // Stream<String>
    .collect(Collectors.toList());
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles");                       // OK
```

Collecting to a `Set`

The method `toSet()` creates a collector that uses a mutable container of type `Set` to perform mutable reduction. The collector does not guarantee to preserve the encounter order of the input stream. For more control over the type of the set, the `toCollection()` method can be used.

The following stream pipeline creates a set with the titles of all CDs in the stream.

Click here to view code image

```
Set<String> cdTitles2 = CD.cdList.stream()        // Stream<CD>
    .map(CD::title)                               // Stream<String>
    .collect(Collectors.toSet());
//[Hot Generics, Java Jive, Lambda Dancing, Keep on Erasing, Java Jam]
```

Collecting to a `Map`

The method `toMap()` creates a collector that performs mutable reduction to a mutable container of type `Map`.

Click here to view code image

```
static <T,K,U> Collector<T,?,Map<K,U>> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper)

static <T,K,U> Collector<T,?,Map<K,U>> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper,
       BinaryOperator<U>                mergeFunction)

static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper,
       BinaryOperator<U>                mergeFunction,
       Supplier<M>                      mapSupplier)
```

Return a `Collector` that accumulates elements of type `T` into a `Map` whose keys and values are the result of applying the provided key and value mapping functions to the input elements.

The `keyMapper` function produces keys of type `K`, and the `valueMapper` function produces values of type `U`.

In the first method, the mapped keys cannot have duplicates—an `Illegal-StateException` will be thrown if that is the case.

In the second and third methods, the `mergeFunction` binary operator is used to resolve collisions between values associated with the same key, as supplied to `Map.merge(Object, Object, BiFunction)`.

In the third method, the provided `mapSupplier` function

> returns a new `Map` into which the results will be inserted.

The collector returned by the method `toMap()` uses either a default map or one that is supplied. To be able to create an entry in a `Map<K,U>` from stream elements of type `T`, the collector requires two functions:

- `keyMapper: T -> K`, which is a `Function` to extract a key of type `K` from a stream element of type `T`.
- `valueMapper: T -> U`, which is a `Function` to extract a value of type `U` for a given key of type `K` from a stream element of type `T`.

Additional functions as arguments allow various controls to be exercised on the map:

- `mergeFunction: (U,U) -> U`, which is a `BinaryOperator` to merge two values that are associated with the same key. The merge function must be specified if collision of values can occur during the mutable reduction, or a resounding exception will be thrown.
- `mapSupplier: () -> M extends Map<K,V>`, which is a `Supplier` that creates a map instance of a specific type to use for mutable reduction. The map created is a subtype of `Map<K,V>`. Without this function, the collector uses a default map.

Figure 16.15 illustrates collecting to a map. The stream pipeline creates a map of CD titles and their release year—that is, a `Map<String, Year>`, where `K` is `String` and `V` is `Year`. The `keyMapper` `CD::title` and the `valueMapper` `CD::year` extract the title (`String`) and the year (`Year`) from each CD in the stream, respectively. The entries are accumulated in a default map (`Map<String, Year>`).

Figure 16.15 *Collecting to a Map*

What if we wanted to create a map with CDs and their release year—that is, a `Map<CD, Year>`? In that case, the `keyMapper` should return the CD as the key—that is, map a CD to itself. That is exactly what the `keyMapper` `Function.identity()` does in the pipeline below.

Click here to view code image

```
Map<CD, Year> mapCDToYear = CD.cdList.stream()
```

```
      .collect(Collectors.toMap(Function.identity(), CD::year)); // Map<CD, Year
```

As there were no duplicates of the key in the previous two examples, there was no collision of values in the map. In the list `dupList` below, there are duplicates of CDs (`CD.cd0`, `CD.cd1`). Executing the pipeline results in a runtime exception at (1).

Click here to view code image

```
List<CD> dupList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd0, CD.cd1);
Map<String, Year> mapTitleToYear1 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year));       // (1)
// IllegalStateException: Duplicate key 2017
```

The collision values can be resolved by specifying a merge function. In the pipeline below, the arguments of the merge function `(y1, y2) -> y1` at (1) have the same value for the year if we assume that a CD can only be released once. Note that `y1` and `y2` denote the existing value in the map and the value to merge, respectively. The merge function can return any one of the values to resolve the collision.

Click here to view code image

```
Map<String, Year> mapTitleToYear2 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year, (y1, y2) -> y1));       //
```

The stream pipeline below creates a map of CD titles released each year. As more than one CD can be released in a year, collision of titles can occur for a year. The merge function `(tt, t) -> tt + ":" + t` concatenates the titles in each year separated by a colon, if necessary. Note that `tt` and `t` denote the existing value in the map and the value to merge, respectively.

Click here to view code image

```
Map<Year, String> mapTitleToYear3 = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
                              (tt, t) -> tt + ":" + t));
//{2017=Java Jive:Java Jam, 2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

The stream pipeline below creates a map with the longest title released each year. For greater control over the type of the map in which to accumulate the entries, a supplier is specified. The supplier `TreeMap::new` returns an empty instance of a `TreeMap` in which the entries are accumulated. The keys in such a map are sorted in their natural order—the class `java.time.Year` implements the `Comparable<Year>` interface.

Click here to view code image

```
TreeMap<Year, String> mapYearToLongestTitle = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
                              BinaryOperator.maxBy(Comparator.naturalOrder()),
                              TreeMap::new));
//{2017=Java Jive, 2018=Lambda Dancing}
```

The merge function specified is equivalent to the following lambda expression, returning the greater of two strings:

Click here to view code image

```
(str1, str2) -> str1.compareTo(str2) > 0 ? str1 : str2
```

Collecting to a `ConcurrentMap`

If the collector returned by the `Collectors.toMap()` method is used in a parallel stream, the multiple partial maps created during parallel execution are merged by the collector to create the final result map. Merging maps can be expensive if keys from one map are merged into another. To address the problem, the `Collectors` class provides the three overloaded methods `toConcurrentMap()`, analogous to the three `toMap()` methods, that return a *concurrent collector*—that is, a collector that uses a single *concurrent map* to perform the reduction. A concurrent map is *thread-safe* and *unordered*. A concurrent map implements the `java.util.concurrent.Concur-rentMap` interface, which is a subinterface of `java.util.Map` interface (§23.7, p. 1482).

Using a concurrent map avoids merging of maps during parallel execution, as a single map is created that is used concurrently to accumulate the results from the execution of each substream. However, the concurrent map is unordered—any encounter order in the stream is ignored. Usage of the `toConcurrentMap()` method is illustrated by the following example of a parallel stream to create a concurrent map of CD titles released each year.

Click here to view code image

```
ConcurrentMap<Year, String> concMapYearToTitles = CD.cdList
    .parallelStream()
    .collect(Collectors.toConcurrentMap(CD::year, CD::title,
                                        (tt, t) -> tt + ":" + t));
//{2017=Java Jam:Java Jive, 2018=Lambda Dancing:Hot Generics:Keep on Erasing}
```

Joining

The `joining()` method creates a collector for concatenating the input elements of type `CharSequence` to a single immutable `String`.

However, internally it uses a mutable `StringBuilder`. Note that the collector returned by the `joining()` methods performs *functional* reduction, as its result is a single immutable string.

> Click here to view code image
>
> ```
> static Collector<CharSequence,?,String> joining()
> static Collector<CharSequence,?,String> joining(CharSequence delimiter)
> static Collector<CharSequence,?,String> joining(CharSequence delimiter,
>                                                 CharSequence prefix,
>                                                 CharSequence suffix)
> ```
>
> Return a `Collector` that concatenates `CharSequence` elements into a `String`. The first method concatenates in encounter order. So does the second method, but this method separates the elements by the specified `delimiter`. The third method in addition applies the specified `prefix` and `suffix` to the result of the concatenation.
>
> The wildcard `?` is a type parameter that is used internally by the collector.
>
> The methods preserve the encounter order, if the stream has one.
>
> Among the classes that implement the `CharSequence` interface are the `String`, `StringBuffer`, and `StringBuilder` classes.

The stream pipelines below concatenate CD titles to illustrate the three overloaded `joining()` methods. The `CharSequence` elements are `String`s. The strings are concatenated in the stream encounter order, which is the positional order for lists. The zero-argument `joining()` method at (1) performs string concatenation of the CD titles using a `StringBuilder` internally, and returns the result as a string.

Click here to view code image

```
String concatTitles1 = CD.cdList.stream()           // Stream<CD>
    .map(CD::title)                                 // Stream<String>
    .collect(Collectors.joining());                 // (1)
//Java JiveJava JamLambda DancingKeep on ErasingHot Generics
```

The single-argument `joining()` method at (2) concatenates the titles using the specified delimiter.

Click here to view code image

```
String concatTitles2 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", "));             // (2) Delimiter
//Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics
```

The three-argument `joining()` method at (3) concatenates the titles using the specified delimiter, prefix, and suffix.

Click here to view code image

```
String concatTitles3 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", ", "[", "]"));  // (3) Delimiter, Prefix, S
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

Grouping

Classifying elements into groups based on some criteria is a very common operation. An example is classifying CDs into groups according to the number of tracks on them (this sounds esoteric, but it will illustrate the point). Such an operation can be accomplished by the collector returned by the `groupingBy()` method. The method is passed a *classifier function* that is used to classify the elements into different groups. The result of the operation is a *classification map* whose entries are the different groups into which the elements have been classified. The key in a map entry is the result of applying the classifier function on the element. The key is extracted from the element based on some property of the element—for example, the number of tracks on the CD. The value associated with a key in a map entry comprises those elements that belong to the same group. The operation is analogous to the *group-by* operation in databases.

There are three versions of the `groupingBy()` method that provide increasingly more control over the grouping operation.

Click here to view code image

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
       Function<? super T,? extends K> classifier)

static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
       Function<? super T,? extends K> classifier,
       Collector<? super T,A,D>        downstream)

static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(
       Function<? super T,? extends K> classifier,
       Supplier<M>                     mapSupplier,
       Collector<? super T,A,D>        downstream)
```

The `Collector` returned by the `groupingBy()` methods implements a *group-by* operation on input elements to create a *classification map*.
The *classifier function* maps elements of type `T` to keys of some type `K`. These keys determine the groups in the classification map.

> The collector returned by the single-argument method produces a classification map of type `Map<K, List<T>>`. The keys in this map are the results from applying the specified classifier function to the input elements. The input elements that map to the same key are accumulated into a `List` by the *default downstream collector* `Collector.toList()`.
>
> The two-argument method accepts a downstream collector, in addition to the classifier function. The collector returned by the method is composed with the specified downstream collector that performs a reduction operation on the input elements that map to the same key. It operates on elements of type `T` and produces a result of type `D`. The result of type `D` produced by the downstream collector is the value associated with the key of type `K`. The composed collector thus results in a classification map of type `Map<K, D>`.
>
> The three-argument method accepts a map supplier as its second parameter. It creates an empty classification map of type `M` that is used by the composed collector. The result is a classification map of type `M` whose key and value types are `K` and `D`, respectively.

Figure 16.16 illustrates the `groupingBy()` operation by grouping CDs according to the number of tracks on them. The classifier function `CD::noOfTracks` extracts the number of tracks from a CD that acts as a key in the classification map (`Map<Integer, List<CD>>`). Since the call to the `groupingBy()` method in Figure 16.16 does not specify a downstream collector, the default downstream collector `Collector.to-List()` is used to accumulate CDs that have the same number of tracks. The number of groups—that is, the number of distinct keys—is equal to the number of distinct values for the number of tracks on the CDs. Each distinct value for the number of tracks is associated with the list of CDs having that value as the number of tracks.

Figure 16.16 *Grouping*

The three stream pipelines below result in a classification map that is equivalent to the one in Figure 16.16. The call to the `groupingBy()` method at (2) specifies the downstream collector explicitly, and is equivalent to the call in Figure 16.16.

Click here to view code image

```
Map<Integer, List<CD>> map22 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList()));  //
```

The call to the `groupingBy()` method at (3) specifies the supplier `TreeMap::new` so that a `TreeMap<Integer, List<CD>>` is used as the classification map.

Click here to view code image

```
Map<Integer, List<CD>> map33 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,                         //
                                   TreeMap::new,
                                   Collectors.toList()));
```

The call to the `groupingBy()` method at (4) specifies the downstream collector `Collector.toSet()` that uses a set to accumulate the CDs for a group.

Click here to view code image

```
Map<Integer, Set<CD>> map44 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toSet()));   //
```

The classification maps created by the pipelines above will contain the three entries shown below, but only the `groupingBy()` method call at (3) can guarantee that the entries will be sorted in a `TreeMap<Integer, List<CD>>` according to the natural order for the `Integer` keys.

Click here to view code image

```
{
6=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
8=[<Jaav, "Java Jive", 8, 2017, POP>,
   <Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
10=[<Funkies, "Lambda Dancing", 10, 2018, POP>,
    <Genericos, "Hot Generics", 10, 2018, JAZZ>]
}
```

In general, any collector can be passed as a downstream collector to the `groupingBy()` method. In the stream pipeline below, the map value in the classification map is a count of the number of CDs having the same number of tracks. The collector `Collector.counting()` performs a functional reduction to count the CDs having the same number of tracks (p. 998).

Click here to view code image

```
Map<Integer, Long> map55 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.counting()));
//{6=1, 8=2, 10=2}
```

Multilevel Grouping

The downstream collector in a `groupingBy()` operation can be created by another `groupingBy()` operation, resulting in a *multilevel grouping* operation—also known as a *multilevel classification* or *cascaded grouping* operation. We can extend the multilevel `groupingBy()` operation to any number of levels by making the downstream collector be a `groupingBy()` operation.

The stream pipeline below creates a classification map in which the CDs are first grouped by the number of tracks in a CD at (1), and then grouped by the musical genre of a CD at (2).

Click here to view code image

```
Map<Integer, Map<Genre, List<CD>>> twoLevelGrp = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,             // (1)
                 Collectors.groupingBy(CD::genre)));           // (2)
```

Printing the contents of the resulting classification map would show the following three entries, not necessarily in this order:

Click here to view code image

```
{
6={JAZZ=[<Jaav, "Java Jam", 6, 2017, JAZZ>]},
8={JAZZ=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
   POP=[<Jaav, "Java Jive", 8, 2017, POP>]},
10={JAZZ=[<Genericos, "Hot Generics", 10, 2018, JAZZ>],
    POP=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
}
```

The entries of the resulting classification map can also be illustrated as a two-dimensional matrix, as shown in Figure 16.16, where the CDs are first grouped into rows by the number of tracks, and then grouped into columns by the musical genre. The value of an element in the matrix is a list of CDs which have the same number of tracks (row) and the same musical genre (column).

Figure 16.17 *Multilevel Grouping as a Two-Dimensional Matrix*

The number of groups in the classification map returned by the above pipeline is equal to the number of distinct values for the number of tracks, as in the single-level `groupingBy()` operation. However, each value associated with a key in the outer classification map is now an inner classification map that is managed by the second-level `groupingBy()` operation. The inner classification map has the type `Map<Genre, List<CD>>`; in other words, the key in the inner classification map is the musical genre of the CD and the value associated with this key is a `List` of CDs with this musical genre. It is the second-level `groupingBy()` operation that is responsible for grouping each CD in the inner classification map. Since no explicit downstream collector is specified for the second-level `groupingBy()` operation, it uses the default downstream collector `Collector.toList()`.

We can modify the multilevel `groupingBy()` operation to count the CDs that have the same musical genre and the same number of tracks by specifying an explicit downstream collector for the second-level `groupingBy()` operation, as shown at (3).

The collector `Collectors.counting()` at (3) performs a functional reduction by accumulating the count for CDs with the same number of tracks and the same musical genre in the inner classification map (p. 998).

Click here to view code image

```
Map<Integer, Map<Genre, Long>> twoLevelGrp2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,
                 Collectors.groupingBy(CD::genre,
                                      Collectors.counting())));   // (3)
```

Printing the contents of the resulting classification map produced by this multilevel `groupingBy()` operation would show the following three entries, again not necessarily in this order:

Click here to view code image

```
{6={JAZZ=1}, 8={JAZZ=1, POP=1}, 10={JAZZ=1, POP=1}}
```

It is instructive to compare the entries in the resulting classification maps in the two examples illustrated here.

To truly appreciate the `groupingBy()` operation, the reader is highly encouraged to implement the multilevel grouping examples in an imperative style, without using the Stream API. Good luck!

Grouping to a `ConcurrentMap`

If the collector returned by the `Collectors.groupingBy()` method is used in a parallel stream, the partial maps created during execution are merged to create the final map—as in the case of the `Collectors.toMap()` method (p. 983). Merging maps can carry a performance penalty. The `Collectors` class provides the three `groupingBy-Concurrent()` overloaded methods, analogous to the three `groupingBy()` methods, that return a *concurrent collector*—that is, a collector that uses a single *concurrent map* to perform the reduction. The entries in such a map are unordered. A concurrent map implements the `java.util.concurrent.ConcurrentMap` interface (§23.7, p. 1482).

Usage of the `groupingByConcurrent()` method is illustrated by the following example of a parallel stream to create a concurrent map of the number of CDs that have the same number of tracks.

Click here to view code image

```
ConcurrentMap<Integer, Long> map66 = CD.cdList
    .parallelStream()
    .collect(Collectors.groupingByConcurrent(CD::noOfTracks,
                                             Collectors.counting()));
//{6=1, 8=2, 10=2}
```

Partitioning

Partitioning is a special case of grouping. The classifier function that was used for grouping is now a *partitioning predicate* in the `partitioningBy()` method. The predicate function returns the `boolean` value `true` or `false`. As the keys of the resulting map are determined by the classifier function, the keys are determined by the partitioning predicate in the case of partitioning. Thus the keys are always of type `Boolean`, implying that the classification map can have, at most, two map entries. In other words, the `partitioningBy()` method can only create, at most, two partitions from the input elements. The map value associated with a key in the resulting map is managed by a downstream collector, as in the case of the `groupingBy()` method.

There are two versions of the `partitioningBy()` method:

> Click here to view code image
>
> ```
> static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
>        Predicate<? super T>     predicate)
>
> static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
>        Predicate<? super T>     predicate,
>        Collector<? super T,A,D> downstream)
> ```
>
> The collector returned by the first method produces a classification map of type `Map<Boolean, List<T>>`. The keys in this map are the results from applying the partitioning predicate to the input elements. The input elements that map to the same `Boolean` key are accumulated into a `List` by the *default downstream collector* `Collector.toList()`.
>
> The second method accepts a downstream collector, in addition to the partitioning predicate. The collector returned by the method is composed with the specified downstream

> collector that performs a reduction operation on the input elements that map to the same key. It operates on elements of type `T` and produces a result of type `D`. The result of type `D` produced by the downstream collector is the value associated with the key of type `Boolean`. The composed collector thus results in a resulting map of type `Map<Boolean, D>`.

Figure 16.18 illustrates the `partitioningBy()` operation by partitioning CDs according to the predicate `CD::isPop` that determines whether a CD is a pop music CD. The result of the partitioning predicate acts as the key in the resulting map of type `Map<Boolean, List<CD>>`. Since the call to the `partitioningBy()` method in Figure 16.18 does not specify a downstream collector, the default downstream collector `Collector.toList()` is used to accumulate CDs that map to the same key. The resulting map has two entries or partitions: one for CDs that are pop music CDs and one for CDs that are not. The two entries of the resulting map are also shown below:

Figure 16.18 *Partitioning*

Click here to view code image

```
{false=[<Jaav, "Java Jam", 6, 2017, JAZZ>,
        <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Generics", 10, 2018, JAZZ>],
 true=[<Jaav, "Java Jive", 8, 2017, POP>,
       <Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

The values in a partition can be obtained by calling the `Map.get()` method:

Click here to view code image

```
List<CD> popCDs = map1.get(true);
List<CD> nonPopCDs = map1.get(false);
```

The stream pipeline at (2) is equivalent to the one in Figure 16.18, where the downstream collector is specified explicitly.

Click here to view code image

```
Map<Boolean, List<CD>> map2 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList())); // (2
```

We could have composed a stream pipeline to filter the CDs that are pop music CDs and collected them into a list. We would have to compose a second pipeline to find the CDs that are *not* pop music CDs. However, the `partitioningBy()` method does both in a single operation.

Analogous to the `groupingBy()` method, any collector can be passed as a downstream collector to the `partitioningBy()` method. In the stream pipeline below, the downstream collector `Collector.counting()` performs a functional reduction to count the number of CDs associated with a key (p. 998).

Click here to view code image

```
Map<Boolean, Long> map3 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.counting()));
//{false=3, true=2}
```

Multilevel Partitioning

Like the `groupingBy()` method, the `partitioningBy()` operation can be used in multilevel classification. The downstream collector in a `partitioningBy()` operation can be created by another `partitioningBy()` operation, resulting in a *multilevel partitioning* operation—also known as a *cascaded partitioning* operation. The downstream collector can also be a `groupingBy()` operation.

In the stream pipeline below, the CDs are partitioned at (1): one partition for CDs that are pop music CDs, and one for those that are not. The CDs that are associated with a key are grouped by the year in which they were released. Note that the CDs that were released in a year are accumulated into a `List` by the default downstream collector `Collector.toList()` that is employed by the `groupingBy()` operation at (2).

Click here to view code image

```
Map<Boolean, Map<Year, List<CD>>> map1 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop,                        // (1)
                 Collectors.groupingBy(CD::year)));                      // (2)
```

Printing the contents of the resulting map would show the following two entries, not necessarily in this order.

Click here to view code image

```
{false={2017=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
        2018=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
              <Genericos, "Hot Generics", 10, 2018, JAZZ>]},
 true={2017=[<Jaav, "Java Jive", 8, 2017, POP>],
       2018=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}}
```

Filtering Adapter for Downstream Collectors

The `filtering()` method of the `Collectors` class encapsulates a *predicate* and a *downstream collector* to create an *adapter* for a *filtering* operation. (See also the `filter()` intermediate operation, p. 912.)

> Click here to view code image
>
> ```
> static <T,A,R> Collector<T,?,R> filtering(
>        Predicate<? super T>     predicate,
>        Collector<? super T,A,R> downstream)
> ```
>
> Returns a `Collector` that applies the `predicate` to input elements of type `T` to determine which elements should be passed to the `downstream` collector. This `downstream` collector accumulates them into results of type `R`, where the type parameter `A` is the intermediate accumulation type of the `downstream` collector.

The following code uses the `filtering()` operation at (2) to group pop music CDs according to the number of tracks on them. The `groupingBy()` operation at (1) creates the groups based on the number of tracks on the CDs, but the `filtering()` operation only allows pop music CDs to pass downstream to be accumulated.

Click here to view code image

```
// Filtering downstream from grouping.
Map<Integer, List<CD>> grpByTracksFilterByPopCD = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,                        // (
                 Collectors.filtering(CD::isPop, Collectors.toList())));  // (
```

Printing the contents of the resulting map would show the entries below, not necessarily in this order. Note that the output shows that there was one or more CDs with six tracks, but there were no pop music CDs. Hence the list of CDs associated with key 6 is empty.

Click here to view code image

```
{6=[],
 8=[<Jaav, "Java Jive", 8, 2017, POP>],
 10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

However, if we run the same query using the `filter()` intermediate stream operation at (1) prior to grouping, the contents of the result map are different, as shown below.

Click here to view code image

```
// Filtering before grouping.
Map<Integer, List<CD>> filterByPopCDGrpByTracks = CD.cdList.stream()
    .filter(CD::isPop)                                                     //
```

```
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList()));
```

Contents of the result map show that only entries that have a non-empty list as a value are contained in the map. This is not surprising, as any non-pop music CD is discarded before grouping, so only pop music CDs are grouped.

Click here to view code image

```
{8=[<Jaav, "Java Jive", 8, 2017, POP>],
 10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

There are no surprises with partitioning, regardless of whether filtering is done before or after the partitioning, as partitioning always results in a map with two entries: one for the `Boolean.TRUE` key and one for the `Boolean.FALSE` key. The code below partitions CDs released in 2018 according to whether a CD is a pop music CD or not.

Click here to view code image

```
// Filtering downstream from partitioning.
Map<Boolean, List<CD>> partbyPopCDsFilterByYear = CD.cdList.stream()     // (1
    .collect(Collectors.partitioningBy(CD::isPop,
                 Collectors.filtering(cd -> cd.year().equals(Year.of(2018)),
                                      Collectors.toList()))); // (2)
// Filtering before partitioning.
Map<Boolean, List<CD>> filterByYearPartbyPopCDs = CD.cdList.stream()     // (2
    .filter(cd -> cd.year().equals(Year.of(2018)))
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList()));
```

Both queries at (1) and (2) above will result in the same entries in the result map:

Click here to view code image

```
{false=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Generics", 10, 2018, JAZZ>],
 true=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

Mapping Adapter for Downstream Collectors

The `mapping()` method of the `Collectors` class encapsulates a *mapper function* and a *downstream collector* to create an *adapter* for a *mapping* operation. (See also the `map()` intermediate operation, p. 921.)

Click here to view code image

```
static <T,U,A,R> Collector<T,?,R> mapping(
       Function<? super T,? extends U> mapper,
       Collector<? super U,A,R>        downstream)
```

> Returns a `Collector` that applies the `mapper` function to input elements of type `T` and provides the mapped results of type `U` to the `downstream` collector that accumulates them into results of type `R`.
>
> In other words, the method adapts a `downstream` collector accepting elements of type `U` to one accepting elements of type `T` by applying a `mapper` function to each input element before accumulation, where type parameter `A` is the intermediate accumulation type of the `downstream` collector.

The `mapping()` method at (1) creates an adapter that accumulates a set of CD titles in each year for a stream of CDs. The mapper function maps a CD to its title so that the downstream collector can accumulate the titles in a set.

Click here to view code image

```
Map<Year, Set<String>> titlesByYearInSet = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                                  // (1)
            CD::title,                                       // Mapper
            Collectors.toSet())));                           // Downstream collector
System.out.println(titlesByYearInSet);
// {2017=[Java Jive, Java Jam],
//  2018=[Hot Generics, Lambda Dancing, Keep on Erasing]}
```

The `mapping()` method at (2) creates an adapter that joins CD titles in each year for a stream of CDs. The mapper function maps a CD to its title so that the downstream collector can join the titles.

Click here to view code image

```
Map<Year, String> joinTitlesByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                                  // (2)
            CD::title,
            Collectors.joining(":"))));
System.out.println(joinTitlesByYear);
// {2017=Java Jive:Java Jam,
//  2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

The `mapping()` method at (3) creates an adapter that counts the number of CD tracks for each year for a stream of CDs. The mapper function maps a CD to its number of tracks so that the downstream collector can count the total number of tracks.

Click here to view code image

```
Map<Year, Long> TotalNumOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
       CD::year,
       Collectors.mapping(                            // (3)
           CD::noOfTracks,
           Collectors.counting())));
System.out.println(TotalNumOfTracksByYear);           // {2017=2, 2018=3}
```

Flat Mapping Adapter for Downstream Collectors

The `flatMapping()` method of the `Collectors` class encapsulates a *mapper function* and a *downstream collector* to create an *adapter* for a *flat mapping* operation. (See also the `flatMap()` intermediate operation, p. 924.)

> Click here to view code image
>
> ```
> static <T,U,A,R> Collector<T,?,R> flatMapping(
>        Function<? super T,? extends Stream<? extends U>> mapper,
>        Collector<? super U,A,R>                          downstream)
> ```
>
> Returns a `Collector` that applies the specified `mapper` function to input elements of type `T` and provides the mapped results of type `U` to the `downstream` collector that accumulates them into results of type `R`.
>
> That is, the method adapts a `downstream` collector accepting elements of type `U` to one accepting elements of type `T` by applying a flat mapping function to each input element before accumulation, where type parameter `A` is the intermediate accumulation type of the `downstream` collector.
>
> The flat mapping function maps an input element to a *mapped stream* whose elements are *flattened* (p. 924) and passed downstream. Each mapped stream is closed after its elements have been flattened. An empty stream is substituted if the mapped stream is `null`.

Given the lists of CDs below, we wish to find all unique CD titles in the lists. A stream of CD lists is created at (1). Each CD list is used to create a stream of CDs whose elements are flattened into the output stream of CDs at (2). Each CD is then mapped to its title at (3), and unique CD titles are accumulated into a set at (4). (Compare this example with the one in Figure 16.9, p. 925, using the `flatMap()` stream operation.)

Click here to view code image

```
// Given lists of CDs:
List<CD> cdListA = List.of(CD.cd0, CD.cd1);
```

```
List<CD> cdListB = List.of(CD.cd0, CD.cd1, CD.cd1);

// Find unique CD titles in the given lists:
Set<String> set =
  Stream.of(cdListA, cdListB)                           // (1) Stream<List<CD>>
        .collect(Collectors.flatMapping(List::stream, // (2) Flatten to Stream
             Collectors.mapping(CD::title,             // (3) Stream<String>
                 Collectors.toSet())));                // (4) Set<String>
```

Set of unique CD titles in the CD lists:

```
[Java Jive, Java Jam]
```

The collectors returned by the `flatMapping()` method are designed to be used in multilevel grouping operations (p. 987), such as downstream from `groupingBy()` or `partitionBy()` operations. Example 16.13 illustrates such a use with the `groupingBy()` operation.

In Example 16.13, the class `RadioPlaylist` at (1) represents a radio station by its name and a list of CDs played by the radio station. Three CD lists are constructed at (2) and used to construct three radio playlists at (3). The radio playlists are stored in a common list of radio playlists at (4). A query is formulated at (5) to find the unique titles of CDs played by each radio station. Referring to the line numbers in Example 16.13:

(6) A stream of type `Stream<RadioPlaylist>` is created from the list `radioPlaylists` of type `RadioPlaylist`.

(7) The radio playlists are grouped according to the name of the radio station (`String`).

(8) Each radio playlist of type `RadioPlaylist` is used as the source of a stream, which is then flattened into the output stream of type `Stream<CD>` by the `flatMapping()` operation.

(9) Each CD in the stream is mapped to its title.

(10) Each unique CD title is accumulated into the result set of each radio station (`Set<String>`).

The query at (5) uses four collectors. The result map has the type `Map<String, List<String>>`. The output shows the unique titles of CDs played by each radio station.

Example 16.13 *Flat mapping*

Click here to view code image

```
import java.util.List;
```

```
// Radio station with a playlist.
public class RadioPlaylist {                                                      //
  private String radioStationName;
  private List<CD> Playlist;

  public RadioPlaylist(String radioStationName, List<CD> cdList) {
    this.radioStationName = radioStationName;
    this.Playlist = cdList;
  }

  public String getRadioStationName() { return this.radioStationName; }
  public List<CD> getPlaylist() { return this.Playlist; }
}
```

Click here to view code image

```
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectorsFlatMapping {
  public static void main(String[] args) {
    // Some lists of CDs:
    List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1, CD.cd2);
    List<CD> cdList2 = List.of(CD.cd0, CD.cd0, CD.cd3);
    List<CD> cdList3 = List.of(CD.cd0, CD.cd4);

    // Some radio playlists:
    RadioPlaylist pl1 = new RadioPlaylist("Radio JVM", cdList1);
    RadioPlaylist pl2 = new RadioPlaylist("Radio JRE", cdList2);
    RadioPlaylist pl3 = new RadioPlaylist("Radio JAR", cdList3);

    // List of radio playlists:
    List<RadioPlaylist> radioPlaylists = List.of(pl1, pl2, pl3);

    // Map of radio station names and set of CD titles they played:
    Map<String, Set<String>> map = radioPlaylists.stream()                    //
        .collect(Collectors.groupingBy(RadioPlaylist::getRadioStationName,    //
             Collectors.flatMapping(rpl -> rpl.getPlaylist().stream(),        //
                 Collectors.mapping(CD::title,                                //
                     Collectors.toSet()))));                                  //
    System.out.println(map);
  }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
{Radio JAR=[Hot Generics, Java Jive],
 Radio JVM=[Java Jive, Lambda Dancing, Java Jam],
 Radio JRE=[Java Jive, Keep on Erasing]}
```

Finishing Adapter for Downstream Collectors

The `collectingAndThen()` method encapsulates a *downstream collector* and a *finisher function* to allow the result of the collector to be adapted by the finisher function.

> Click here to view code image
>
> ```
> static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(
>        Collector<T,A,R> downstream,
>        Function<R,RR>   finisher)
> ```
>
> Returns a `Collector` that performs the operation of the `downstream` collector on input elements of type `T`, followed by applying the `finisher` function on the result of type `R` produced by the `downstream` collector. The final result is of type `RR`, the result of the finisher function. In other words, the method adapts a collector to perform an additional finishing transformation.

In the call to the `collectAndThen()` method at (1), the collector `Collectors.maxBy()` at (2) produces an `Optional<Integer>` result that is the maximum CD by number of tracks in each group. The finisher function at (3) extracts the value from the `Optional<Integer>` result, if there is one; otherwise, it returns `0`. The `collectAndThen()` method adapts the `Optional<Integer>` result of its argument collector to an `Integer` value by applying the finisher function.

Click here to view code image

```
Map<Year, Integer> maxTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.collectingAndThen(                                  // (1)
             Collectors.maxBy(Comparator.comparing(CD::noOfTracks)),    // (2)
             optCD -> optCD.map(CD::noOfTracks).orElse(0)))             // (3)
     );
System.out.println(maxTracksByYear);                      // {2017=8, 2018=10}
```

In the call to the `collectAndThen()` method at (4), the collector `Collectors.averaging-Double()` at (5) produces a result of type `Double` that is the average number of tracks in each group. The finisher function at (6) maps the `Double` average value to a string with the specified number format.

Click here to view code image

```
Map<Genre, String> avgTracksByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::genre,
         Collectors.collectingAndThen(                                  // (4)
             Collectors.averagingDouble(CD::noOfTracks),                // (5)
             d -> String.format("%.1f", d)))                            // (6)
     );
System.out.println(avgTracksByGenre);                   // {JAZZ=8.0, POP=9.0}
```

Downstream Collectors for Functional Reduction

The collectors we have seen so far perform a *mutable reduction* to some *mutable container*, except for the functional reduction implemented by the `joining()` method (p. 984). The `Collectors` class also provides static factory methods that implement collectors which perform *functional reduction* to compute common statistics, such as summing, averaging, finding maximum and minimum values, and the like.

Like any other collector, the collectors that perform functional reduction can also be used in a standalone capacity as a parameter of the `collect()` method and as a downstream collector in a composed collector. However, these collectors are most useful when used as downstream collectors.

Collectors performing functional reduction have counterparts in terminal operations for streams that provide equivalent reduction operations (Table 16.8, p. 1008).

Counting

The collector created by the `Collectors.counting()` method performs a functional reduction to count the input elements.

> Click here to view code image
>
> ```
> static <T> Collector<T,?,Long> counting()
> ```
>
> The collector returned counts the number of input elements of type `T`. If there are no elements, the result is `Long.valueOf(0L)`. Note that the result is of type `Long`.
>
> The wildcard `?` represents any type, and in the method declaration, it is the type parameter for the mutable type that is accumulated by the reduction operation.

In the stream pipeline at (1), the collector `Collectors.counting()` is used in a standalone capacity to count the number of jazz music CDs.

Click here to view code image

```
Long numOfJazzCds1 = CD.cdList.stream().filter(CD::isJazz)
    .collect(Collectors.counting());                    // (1) Standalone collec
System.out.println(numOfJazzCds1);                      // 3
```

In the stream pipeline at (2), the collector `Collectors.counting()`

is used as a downstream collector in a grouping operation that groups the CDs by musical genre and uses the downstream collector to count the number of CDs in each group.

Click here to view code image

```
Map<Genre, Long> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
                 CD::genre,
                 Collectors.counting()));             // (2) Downstream collec
System.out.println(grpByGenre);                       // {POP=2, JAZZ=3}
System.out.println(grpByGenre.get(Genre.JAZZ));       // 3
```

The collector `Collectors.counting()` performs effectively the same functional reduction as the `Stream.count()` terminal operation (p. 953) at (3).

Click here to view code image

```
long numOfJazzCds2 = CD.cdList.stream().filter(CD::isJazz)
    .count();                                         // (3) Stream.count()
System.out.println(numOfJazzCds2);                    // 3
```

Finding Min/Max

The collectors created by the `Collectors.maxBy()` and `Collectors.minBy()` methods perform a functional reduction to find the maximum and minimum elements in the input elements, respectively. As there might not be any input elements, an `Optional<T>` is returned as the result.

> Click here to view code image
>
> ```
> static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> cmp)
> static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> cmp)
> ```
>
> Return a collector that produces an `Optional<T>` with the maximum or minimum element of type `T` according to the specified `Comparator`, respectively.

The natural order comparator for CDs defined at (1) is used in the stream pipelines below to find the maximum CD. The collector `Collectors.maxBy()` is used as a standalone collector at (2), using the natural order comparator to find the maximum CD. The `Optional<CD>` result can be queried for the value.

Click here to view code image

```
Comparator<CD> natCmp = Comparator.naturalOrder(); // (1)

Optional<CD> maxCD = CD.cdList.stream()
```

```
    .collect(Collectors.maxBy(natCmp));             // (2) Standalone collector
System.out.println("Max CD: "
    + maxCD.map(CD::title).orElse("No CD."));      // Max CD: Java Jive
```

In the pipeline below, the CDs are grouped by musical genre, and the CDs in each group are reduced to the maximum CD by the downstream collector `Collectors.maxBy()` at (3). Again, the downstream collector uses the natural order comparator, and the `Optional<CD>` result in each group can be queried.

Click here to view code image

```
// Group CDs by musical genre, and max CD in each group.
Map<Genre, Optional<CD>> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.maxBy(natCmp)));       // (3) Downstream collector
System.out.println(grpByGenre);
//{JAZZ=Optional[<Jaav, "Java Jam", 6, 2017, JAZZ>],
// POP=Optional[<Jaav, "Java Jive", 8, 2017, POP>]}

System.out.println("Title of max Jazz CD: "
    + grpByGenre.get(Genre.JAZZ)
                .map(CD::title)
                .orElse("No CD."));       // Title of max Jazz CD: Java Jam
```

The collectors created by the `Collectors.maxBy()` and `Collectors.minBy()` methods are effectively equivalent to the `max()` and `min()` terminal operations provided by the stream interfaces (p. 954), respectively. In the pipeline below, the `max()` terminal operation reduces the stream of `CD`s to the maximum CD at (4) using the natural order comparator, and the `Optional<CD>` result can be queried.

Click here to view code image

```
Optional<CD> maxCD1 = CD.cdList.stream()
    .max(natCmp);                         // (4) max() on Stream<CD>.
System.out.println("Title of max CD: "
    + maxCD1.map(CD::title)
            .orElse("No CD."));           // Title of max CD: Java Jive
```

Summing

The summing collectors perform a functional reduction to produce the sum of the numeric results from applying a numeric-valued function to the input elements.

Click here to view code image

```
static <T> Collector<T,?,NumType> summingNumType(
       ToNumTypeFunction<? super T> mapper)
```

> Returns a collector that produces the sum of a *`numtype`*-valued function applied to the input elements. If there are no input elements, the result is zero. The result is of *`NumType`*.
>
> *NumType* is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding *`numtype`* is `int`, `long`, or `double`.

The collector returned by the `Collectors.summingInt()` method is used at (1) as a standalone collector to find the total number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

Click here to view code image

```
Integer sumTracks = CD.cdList.stream()
    .collect(Collectors.summingInt(CD::noOfTracks));   // (1) Standalone colle
System.out.println(sumTracks);                         // 42
```

In the pipeline below, the CDs are grouped by musical genre, and the number of tracks on CDs in each group summed by the downstream collector is returned by the `Collectors.summingInt()` method at (2).

Click here to view code image

```
Map<Genre, Integer> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::genre,
         Collectors.summingInt(CD::noOfTracks)));    // (2) Downstream collect
System.out.println(grpByGenre);                      // {POP=18, JAZZ=24}
System.out.println(grpByGenre.get(Genre.JAZZ));      // 24
```

The collector `Collectors.summingInt()` performs effectively the same functional reduction at (3) as the `IntStream.sum()` terminal operation (p. 973).

Click here to view code image

```
int sumTracks2 = CD.cdList.stream()                  // (3) Stream<CD>
    .mapToInt(CD::noOfTracks)                        // IntStream
    .sum();
System.out.println(sumTracks2);                      // 42
```

Averaging

The averaging collectors perform a functional reduction to produce the average of the numeric results from applying a numeric-valued function to the input elements.

> Click here to view code image
>
> ```
> static <T> Collector<T,?,Double> averagingNumType(
>        ToNumTypeFunction<? super T> mapper)
> ```
>
> Returns a collector that produces the arithmetic mean of a *numtype*-valued function applied to the input elements. If there are no input elements, the result is zero. The result is of type `Double`.
>
> *NumType* is `Int`, `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

The collector returned by the `Collectors.averagingInt()` method is used at (1) as a standalone collector to find the average number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

Click here to view code image

```
Double avgNoOfTracks1 = CD.cdList.stream()
    .collect(Collectors
        .averagingInt(CD::noOfTracks));                 // (1) Standalone collecto
System.out.println(avgNoOfTracks1);                     // 8.4
```

In the pipeline below, the CDs are grouped by musical genre, and the downstream collector `Collectors.averagingInt()` at (2) calculates the average number of tracks on the CDs in each group.

Click here to view code image

```
Map<Genre, Double> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
       CD::genre,
       Collectors.averagingInt(CD::noOfTracks)          // (2) Downstream collecto
       ));
System.out.println(grpByGenre);                         // {POP=9.0, JAZZ=8.0}
System.out.println(grpByGenre.get(Genre.JAZZ));         // 8.0
```

The collector created by the `Collectors.averagingInt()` method performs effectively the same functional reduction as the `IntStream.average()` terminal operation (p. 974) at (3).

Click here to view code image

```
OptionalDouble avgNoOfTracks2 = CD.cdList.stream()    // Stream<CD>
    .mapToInt(CD::noOfTracks)                           // IntStream
    .average();                                         // (3)
System.out.println(avgNoOfTracks2.orElse(0.0));         // 8.4
```

Summarizing

The summarizing collector performs a functional reduction to produce summary statistics (count, sum, min, max, average) on the numeric results of applying a numeric-valued function to the input elements.

> Click here to view code image
>
> ```
> static <T> Collector<T,?,NumTypeSummaryStatistics> summarizingNumType(
>        ToNumTypeFunction<? super T> mapper)
> ```
>
> Returns a collector that applies a *numtype*-valued mapper function to the input elements, and returns the summary statistics for the resulting values.
>
> *NumType* is `Int` (but it is `Integer` when used as a type name), `Long`, or `Double`, and the corresponding *numtype* is `int`, `long`, or `double`.

The collector `Collectors.summarizingInt()` is used at (1) as a standalone collector to summarize the statistics for the number of tracks on the CDs. The mapper function `CD::noOfTracks` passed as an argument extracts the number of tracks from each CD on which the functional reduction is performed.

Click here to view code image

```
IntSummaryStatistics stats1 = CD.cdList.stream()
    .collect(
      Collectors.summarizingInt(CD::noOfTracks)      // (1) Standalone collect
     );
System.out.println(stats1);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

The `IntSummaryStatistics` class provides get methods to access the individual results (p. 974).

In the pipeline below, the CDs are grouped by musical genre, and the downstream collector created by the `Collectors.summarizingInt()` method at (2) summarizes the statistics for the number of tracks on the CDs in each group.

Click here to view code image

```
Map<Genre, IntSummaryStatistics> grpByGenre = CD.cdList.stream()
  .collect(Collectors.groupingBy(
     CD::genre,
     Collectors.summarizingInt(CD::noOfTracks)));    // (2) Downstream collect
System.out.println(grpByGenre);
//{POP=IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10},
// JAZZ=IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}

System.out.println(grpByGenre.get(Genre.JAZZ));   // Summary stats for Jazz CD
// IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

The collector returned by the `Collectors.summarizingInt()` method performs effectively the same functional reduction as the `IntStream.summaryStatistics()` terminal operation (p. 974) at (3).

Click here to view code image

```
IntSummaryStatistics stats2 = CD.cdList.stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();                            // (3)
System.out.println(stats2);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

Reducing

Collectors that perform common statistical operations, such as counting, averaging, and so on, are special cases of functional reduction that can be implemented using the `Collectors.reducing()` method.

> Click here to view code image
>
> ```
> static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> bop)
> ```
>
> Returns a collector that performs functional reduction, producing an `Optional` with the *cumulative* result of applying the binary operator `bop` on the input elements: `e1 bop e2 bop e3 ...`, where each `ei` is an input element. If there are no input elements, an empty `Optional<T>` is returned.
>
> Note that the collector reduces input elements of type `T` to a result that is an `Optional` of type `T`.
>
> Click here to view code image
>
> ```
> static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> bop)
> ```
>
> Returns a collector that performs functional reduction, producing the *cumulative* result of applying the binary operator `bop` on the input elements: `identity bop e1 bop e2 ...`, where each `ei` is an input element. The `identity` value is the initial value to accumulate. If there are no input elements, the `identity` value is returned.
>
> Note that the collector reduces input elements of type `T` to a result of type `T`.
>
> Click here to view code image
>
> ```
> static <T,U> Collector<T,?,U> reducing(
>        U                                 identity,
>        Function<? super T,? extends U> mapper,
>        BinaryOperator<U>                 bop)
> ```

> Returns a collector that performs a *map-reduce* operation. It maps each input element of type `T` to a mapped value of type `U` by applying the `mapper` function, and performs functional reduction on the mapped values of type `U` by applying the binary operator `bop`. The `identity` value of type `U` is used as the initial value to accumulate. If the stream is empty, the `identity` value is returned.
>
> Note that the collector reduces input elements of type `T` to a result of type `U`.

Collectors returned by the `Collectors.reducing()` methods effectively perform equivalent functional reductions as the `reduce()` methods of the stream interfaces. However, the three-argument method `Collectors.reducing(identity, mapper, bop)` performs a map-reduce operation using a `mapper` function and a binary operator `bop`, whereas the `Stream.reduce(identity, accumulator, combiner)` performs a reduction using an `accumulator` and a `combiner` (p. 955). The `accumulator` is a `BiFunction<U,T,U>` that accumulates a partially accumulated result of type `U` with an element of type `T`, whereas the `bop` is a `BinaryOperator<U>` that accumulates a partially accumulated result of type `U` with an element of type `U`.

The following comparators are used in the examples below:

Click here to view code image

```
// Comparator to compare CDs by title.
Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);        // (1)
// Comparator to compare strings by their length.
Comparator<String> byLength = Comparator.comparing(String::length); // (2)
```

The collector returned by the `Collectors.reducing()` method is used as a standalone collector at (3) to find the CD with the longest title. The result of the operation is an `Optional<String>` as there might not be any input elements. This operation is equivalent to using the `Stream.reduce()` terminal operation at (4).

Click here to view code image

```
Optional<String> longestTitle1 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.reducing(
        BinaryOperator.maxBy(byLength)));            // (3) Standalone collect
System.out.println(longestTitle1.orElse("No title"));// Keep on Erasing

Optional<String> longestTitle2 = CD.cdList.stream()  // Stream<CD>
    .map(CD::title)                                  // Stream<String>
    .reduce(BinaryOperator.maxBy(byLength));         // (4) Stream.reduce(bop)
```

The collector returned by the one-argument `Collectors.reducing()` method is used as a downstream collector at (5) to find the CD with the longest title in each group classified by the year a CD was released. The collector at (5) is equivalent to the collector returned by the `Collectors.maxBy(cmpByTitle)` method.

Click here to view code image

```
Map<Year, Optional<CD>> cdWithMaxTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.reducing(                           // (5) Downstream collect
             BinaryOperator.maxBy(cmpByTitle))
         ));
System.out.println(cdWithMaxTitleByYear);
// {2017=Optional[<Jaav, "Java Jive", 8, 2017, POP>],
//  2018=Optional[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
System.out.println(cdWithMaxTitleByYear.get(Year.of(2018))
                       .map(CD::title).orElse("No title")); // Lambda Dancing
```

The collector returned by the three-argument `Collectors.reducing()` method is used as a downstream collector at (6) to find the longest title in each group classified by the year a CD was released. Note that the collector maps a CD to its title. The longest title is associated with the map value for each group classified by the year a CD was released. The collector will return an empty string (i.e., the identity value `""`) if there are no CDs in the stream. In comparison, the collector `Collectors.mapping()` at (7) also maps a CD to its title, and uses the downstream collector `Collectors.maxBy(byLength)` at (7) to find the longest title (p. 993). The result in this case is an `Optional<String>`, as there might not be any input elements.

Click here to view code image

```
Map<Year, String> longestTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.reducing("", CD::title,             // (6) Downstream collect
             BinaryOperator.maxBy(byLength))
         ));
System.out.println(longestTitleByYear);  // {2017=Java Jive, 2018=Keep on Eras
System.out.println(longestTitleByYear.get(Year.of(2018)));        // Keep on Era

Map<Year, Optional<String>> longestTitleByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.mapping(CD::title,                  // (7) Downstream collect
             Collectors.maxBy(byLength))
         ));
System.out.println(longestTitleByYear2);
// {2017=Optional[Java Jive], 2018=Optional[Keep on Erasing]}
System.out.println(longestTitleByYear2.get(Year.of(2018))
                       .orElse("No title."));        // Keep on Erasing
```

The pipeline below groups CDs according to the year they were released. For each group, the collector returned by the three-argument `Collectors.reducing()` method performs a map-reduce operation at (8) to map each CD to its number of tracks and accumulate the tracks in each group. This map-reduce operation is equivalent to the collector returned by the `Collectors.summingInt()` method at (9).

Click here to view code image

```
Map<Year, Integer> noOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.reducing(                        // (8) Downstream collect
             0, CD::noOfTracks, Integer::sum)));
System.out.println(noOfTracksByYear);                   // {2017=14, 2018=28}
System.out.println(noOfTracksByYear.get(Year.of(2018)));// 28

Map<Year, Integer> noOfTracksByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.summingInt(CD::noOfTracks)));    // (9) Special case colle
```

Summary of Static Factory Methods in the `Collectors` Class

The static factory methods of the `Collectors` class that create collectors are summarized in Table 16.7. All methods are `static` generic methods, except for the overloaded `joining()` methods that are not generic. The keyword `static` is omitted, as are the type parameters of a generic method, since these type parameters are evident from the declaration of the formal parameters to the method. The type parameter declarations have also been simplified, where any bound `<? super T>` or `<? extends T>` has been replaced by `<T>`, without impacting the intent of a method. A reference is also provided for each method in the first column.

The last column in Table 16.7 indicates the function type of the corresponding parameter in the previous column. It is instructive to note how the functional interface parameters provide the parameterized behavior to build the collector returned by a method. For example, the method `averagingDouble`() returns a collector that computes the average of the stream elements. The parameter function `mapper` with the functional interface type `ToDoubleFunction<T>` converts an element of type `T` to a `double` when the collector computes the average for the stream elements.

Table 16.7 *Static Methods in the* `Collectors` *Class*

Method name (ref. p.)	Return type	Type of parameters	Functional type of parameters
`averagingDouble` (p. 1000)	`Collector<T,?,Double>`	`(ToDoubleFunction<T> mapper)`	`T -> double`
`averagingInt` (p. 1000)	`Collector<T,?,Double>`	`(ToIntFunction<T> mapper)`	`T -> int`
`averagingLong` (p. 1000)	`Collector<T,?,Double>`	`(ToLongFunction<T> mapper)`	`T -> long`
`collectingAndThen` (p. 997)	`Collector<T,A,RR>`	`(Collector<T,A,R> downstream, Function<R,RR> finisher)`	`(T,A) -> R,` `R -> RR`
`counting` (p. 998)	`Collector<T,?,Long>`	`()`	
`filtering` (p. 992)	`Collector<T,?,R>`	`(Predicate<T> predicate, Collector<T,A,R> downstream)`	`T -> boolean,` `(T,A) -> R`
`flatMapping` (p. 994)	`Collector<T,?,R>`	`(Function<T, Stream<U>> mapper, Collector<U,A,R> downstream)`	`T -> Stream<U>,` `(U,A) -> R`
`groupingBy` (p. 985)	`Collector<T,?, Map<K,List<T>>>`	`(Function<T,K> classifier)`	`T -> K`
`groupingBy` (p. 985)	`Collector<T,?, Map<K,D>>`	`(Function<T,K> classifier, Collector<T,A,D> downstream)`	`T -> K,` `(T,A) -> D`

`groupingBy` (p. 985)	`Collector<T,?,Map<K,D>>`	`(Function<T,K> classifier, Supplier<Map<K,D>> mapSupplier, Collector<T,A,D> downstream)`	`T -> K,`
			`()->Map<K,D>,`
			`(T,A)->D`

`joining` (p. 984)	`Collector <CharSequence,?,String>`	`()`	
`joining` (p. 984)	`Collector <CharSequence,?,String>`	`(CharSequence delimiter)`	
`joining` (p. 984)	`Collector <CharSequence,?,String>`	`(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`	
`mapping` (p. 993)	`Collector<T,?,R>`	`(Function<T,U> mapper, Collector<U,A,R> downstream)`	`T -> U,` `(U,A) -> R`
`maxBy` (p. 999)	`Collector<T,?,Optional<T>>`	`(Comparator<T> cmp)`	`(T,T) -> int`
`minBy` (p. 999)	`Collector<T,?,Optional<T>>`	`(Comparator<T> cmp)`	`(T,T) -> int`
`partitioningBy` (p. 989)	`Collector<T,?, Map<Boolean,List<T>>>`	`(Predicate<T> predicate)`	`T -> boolean`
`partitioningBy` (p. 989)	`Collector<T,?, Map<Boolean,D>>`	`(Predicate<T> predicate, Collector<T,A,D> downstream)`	`T -> boolean,` `(T,A) -> D`
`reducing` (p. 1002)	`Collector<T,?,Optional<T>>`	`(BinaryOperator<T> op)`	`(T,T) -> T`
`reducing` (p. 1002)	`Collector<T,?,T>`	`(T identity, BinaryOperator<T> op)`	`(T,T) -> T`
`reducing` (p. 1002)	`Collector<T,?,U>`	`(U identity, Function<T,U> mapper, BinaryOperator<U> op)`	`T -> U,` `(U,U) -> U`

summarizingDouble (p. 1001)	Collector<T,?,DoubleSummaryStatistics>	(ToDoubleFunction<T> mapper)	T->double
summarizingInt (p. 1001)	Collector<T,?,IntSummaryStatistics>	(ToIntFunction<T> mapper)	T->int
summarizingLong (p. 1001)	Collector<T,?,LongSummaryStatistics>	(ToLongFunction<T> mapper)	T->long
summingDouble (p. 978)	Collector<T,?,Double>	(ToDoubleFunction<T> mapper)	T->double
summingInt (p. 978)	Collector<T,?,Integer>	(ToIntFunction<T> mapper)	T->int
summingLong (p. 978)	Collector<T,?,Long>	(ToLongFunction<T> mapper)	T->long
toCollection (p. 979)	Collector<T,?,C>	(Supplier<C> collFactory)	()->C
toList toUnmodifiableList (p. 980)	Collector<T,?,List<T>>	()	
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper)	T->K, T->U
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper, BinaryOperator<U> mergeFunction)	T->K, T->U, (U,U)->U
toMap (p. 981)	Collector<T,?,Map<K,U>>	(Function<T,K> keyMapper, Function<T,U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<Map<K,U>> mapSupplier)	T->K, T->U, (U,U)->U, ()->Map<K,U>
toSet toUnmodifiableSet (p. 980)	Collector<T,?,Set<T>>	()	

Table 16.8 shows a comparison of methods in the stream interfaces that perform reduction operations and static factory methods in the `Collectors` class that implement collectors with equivalent functionality.

Table 16.8 *Method Comparison: The Stream Interfaces and the* `Collectors` *Class*

Method in the stream interfaces	Static factory method in the `Collectors` class
collect (p. 964)	collectingAndThen (p. 997)
count (p. 953)	counting (p. 998)
filter (p. 912)	filtering (p. 992)
flatMap (p. 924)	flatMapping (p. 994)
map (p. 921)	mapping (p. 993)
max (p. 954)	maxBy (p. 999)
min (p. 954)	minBy (p. 999)
reduce (p. 955)	reducing (p. 1002)
toList (p. 972)	toList (p. 980)
average (p. 972)	averagingInt, averagingLong, averagingDouble (p. 1001)
sum (p. 972)	summingInt, summingLong, summingDouble (p. 978)
summaryStatistics (p. 972)	summarizingInt, summarizingLong, summarizingDouble (p. 1001)

16.9 Parallel Streams

The Stream API makes it possible to execute a sequential stream in parallel without rewriting the code. The primary reason for using parallel streams is to improve performance, but at the same time ensuring that the results obtained are the same, or at least compatible, regardless of the mode of execution. Although the API goes a long way to achieve its aim, it is important to understand the pitfalls to avoid when executing stream pipelines in parallel.

Building Parallel Streams

The execution mode of an existing stream can be set to parallel by calling the `parallel()` method on the stream (p. 933). The `parallelStream()` method of the `Collection` interface can be used to create a parallel stream with a collection as the data source (p. 897). No other code is necessary for parallel execution, as the data partitioning and thread management for a parallel stream are handled by the API and the JVM. As with any stream, the stream is not executed until a terminal operation is invoked on it.

The `isParallel()` method of the stream interfaces can be used to determine whether the execution mode of a stream is parallel (p. 933).

Parallel Stream Execution

The Stream API allows a stream to be executed either sequentially or in parallel— meaning that all stream operations can execute either sequentially or in parallel. A sequential stream is executed in a single thread running on one CPU core. The elements in the stream are processed sequentially in a single pass by the stream operations that are executed in the same thread (p. 891).

A parallel stream is executed by different threads, running on multiple CPU cores in a computer. The stream elements are split into substreams that are processed by multiple instances of the stream pipeline being executed in multiple threads. The partial results from processing of each substream are merged (or combined) into a final result (p. 891).

Parallel streams utilize the Fork/Join Framework (§23.3, p. 1447) under the hood for executing parallel tasks. This framework provides support for the thread management necessary to execute the substreams in parallel. The number of threads employed during parallel stream execution is dependent on the CPU cores in the computer.

Figure 16.12, p. 963, illustrates parallel functional reduction using the three-argument `reduce(identity, accumulator, combiner)` terminal operation (p. 962).

Figure 16.14, p. 967, illustrates parallel mutable reduction using the three-argument `collect(supplier, accumulator, combiner)` terminal operation (p. 966).

Factors Affecting Performance

There are no guarantees that executing a stream in parallel will improve the performance. In this subsection we look at some factors that can affect performance.

Benchmarking

In general, increasing the number of CPU cores and thereby the number of threads that can execute in parallel only scales performance up to a threshold for a given size of data, as some threads might become idle if there is no data left for them to process. The number of CPU cores boosts performance to a certain extent, but it is not the only factor that should be considered when deciding to execute a stream in parallel.

Inherent in the total cost of parallel processing is the start-up cost of setting up the parallel execution. At the onset, if this cost is already comparable to the cost of sequential execution, not much can be gained by resorting to parallel execution.

A combination of the following three factors can be crucial in deciding whether a stream should be executed in parallel:

- *Sufficiently large data size*

  The size of the stream must be large enough to warrant parallel processing; otherwise, sequential processing is preferable. The start-up cost can be too prohibitive for parallel execution if the stream size is too small.

- *Computation-intensive stream operations*

  If the stream operations are small computations, then the stream size should be proportionately large as to warrant parallel execution. If the stream operations are computation-intensive, the stream size is less significant, and parallel execution can boost performance.

- *Easily splittable stream*

  If the cost of splitting the stream into substreams is higher than processing the substreams, employing parallel execution can be futile. Collections like `Array-List`s, `HashMap`s, and simple arrays are efficiently splittable, whereas `LinkedList`s and IO-based data sources are less efficient in this regard.

Benchmarking—that is, measuring performance—is strongly recommended to decide whether parallel execution will be beneficial. Example 16.14 illustrates a simple scheme where reading the system clock before and after a stream is executed can be used to get a sense of how well a stream performs.

The class `StreamBenchmarks` in Example 16.14 defines five methods, at (1) through (5), that compute the sum of values from `1` to `n`. These methods compute the sum in various ways. Each method is executed with four different values of `n`; that is, the stream size is the number of values for summation. The program prints the benchmarks for each method for the different values of `n`, which of course can vary, as many factors can influence the results—the most significant one being the number of CPU cores on the computer.

- The methods `seqSumRangeClosed()` at (1) and `parSumRangeClosed()` at (2) perform the computation on a sequential and a parallel stream, respectively, that are created with the `closeRange()` method.

  Click here to view code image

```
return LongStream.rangeClosed(1L, n).sum();                // Sequentia
...
return LongStream.rangeClosed(1L, n).parallel().sum();     // Parallel
```

  Benchmarks from Example 16.14:

```
  Size   Sequential Parallel
   1000   0.05681   0.11031
  10000   0.06698   0.13979
 100000   0.71274   0.52627
1000000   7.02237   4.37249
```

  The terminal operation `sum()` is not computation-intensive. The parallel stream starts to show better performance when the number of values approaches `100000`. The stream size is then significantly large for the parallel stream to show better performance. Note that the range of values defined by the arguments of the `rangeClosed()` method can be efficiently split into substreams, as its start and end values are provided.

- The methods `seqSumIterate()` at (3) and `parSumIterate()` at (4) return a sequential and a parallel sequential stream, respectively, that is created with the `iterate()` method.

  Click here to view code image

  ```
  return LongStream.iterate(1L, i -> i + 1).limit(n).sum();            //
  ...
  return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum(); //
  ```

  Benchmarks from Example 16.14:

  ```
    Size   Sequential Parallel
     1000   0.08645   0.34696
    10000   0.35687   1.27861
   100000   3.24083  11.38709
  1000000  29.92285 117.87909
  ```

  The method `iterate()` creates an infinite stream, and the `limit()` intermediate operation truncates the stream according to the value of `n`. The performance of both streams degrades fast when the number of values increases. However, the parallel stream performs worse than the sequential stream in all cases. The values generated by the `iterate()` method are not known before the stream is executed, and the `limit()` operation is also stateful, making the process of splitting the values into substreams inefficient in the case of the parallel stream.

- The method `iterSumLoop()` at (5) uses a `for(;;)` loop to compute the sum.

  Benchmarks from Example 16.14:

  ```
    Size   Iterative
     1000   0.00586
    10000   0.02330
   100000   0.22352
  1000000   2.49677
  ```

  Using a `for(;;)` loop to calculate the sum performs best for all values of `n` compared to the streams, showing that significant overhead is involved in using streams for summing a sequence of numerical values.

In Example 16.14, the methods `measurePerf()` at (6) and `xqtFunctions()` at (13) create the benchmarks for functions passed as parameters. In the `measurePerf()` method, the system clock is read at (8) and the function parameter `func` is applied at (9). The system clock is read again at (10) after the function application at (9) has completed. The execution time calculated at (10) reflects the time for executing the function. Applying the function `func` evaluates the lambda expression or the method reference implementing the `LongFunction` interface. In Example 16.14, the function parameter `func` is implemented by method references that call methods, at (1) through (5), in the `StreamBenchmarks` class whose execution time we want to measure.

Click here to view code image

```
public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
  // ...
  double start = System.nanoTime();                              // (8)
  result = func.apply(n);                                        // (9)
  double duration = (System.nanoTime() - start)/1_000_000;       // (10) ms.
  // ...
}
```

Example 16.14 *Benchmarking*

Click here to view code image

```
import java.util.function.LongFunction;
import java.util.stream.LongStream;
/*
 * Benchmark the execution time to sum numbers from 1 to n values
 * using streams.
 */
public final class StreamBenchmarks {

  public static long seqSumRangeClosed(long n) {                     // (1)
    return LongStream.rangeClosed(1L, n).sum();
  }

  public static long paraSumRangeClosed(long n) {                    // (2)
    return LongStream.rangeClosed(1L, n).parallel().sum();
  }

  public static long seqSumIterate(long n) {                         // (3)
    return LongStream.iterate(1L, i -> i + 1).limit(n).sum();
  }

  public static long paraSumIterate(long n) {                        // (5)
    return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum();
  }

  public static long iterSumLoop(long n) {                           // (5)
    long result = 0;
    for (long i = 1L; i <= n; i++) {
      result += i;
    }
    return result;
  }
  /*
   * Applies the function parameter func, passing n as parameter.
   * Returns the average time (ms.) to execute the function 100 times.
   */
  public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
    int numOfExecutions = 100;
    double totTime = 0.0;
    R result = null;
    for (int i = 0; i < numOfExecutions; i++) {                        // (7)
      double start = System.nanoTime();                                // (8)
      result = func.apply(n);                                          // (9)
      double duration = (System.nanoTime() - start)/1_000_000;         // (10)
      totTime += duration;                                             // (11)
    }
    double avgTime = totTime/numOfExecutions;                          // (12)
    return avgTime;
  }

  /*
   * Executes the functions in the varargs parameter funcs
   * for different stream sizes.
   */
  public static <R> void xqtFunctions(LongFunction<R>... funcs) {      // (13)
    long[] sizes = {1_000L, 10_000L, 100_000L, 1_000_000L};            // (14)

    // For each stream size ...
    for (int i = 0; i < sizes.length; ++i) {                           // (15)
      System.out.printf("%7d", sizes[i]);
      // ... execute the functions passed in the varargs parameter funcs.
      for (int j = 0; j < funcs.length; ++j) {                         // (16)
        System.out.printf("%10.5f", measurePerf(funcs[j], sizes[i]));
      }
      System.out.println();
    }
  }

  public static void main(String[] args) {                             // (17)

    System.out.println("Streams created with the rangeClosed() method:");// (1
    System.out.println("  Size   Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumRangeClosed,
                 StreamBenchmarks::paraSumRangeClosed);

    System.out.println("Streams created with the iterate() method:");  // (19)
    System.out.println("  Size   Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumIterate,
                 StreamBenchmarks::paraSumIterate);

    System.out.println("Iterative solution with an explicit loop:");   // (20)
    System.out.println("  Size   Iterative");
    xqtFunctions(StreamBenchmarks::iterSumLoop);
  }
}
```

Possible output from the program:

Click here to view code image

```
Streams created with the rangeClosed() method:
  Size   Sequential Parallel
   1000   0.05681   0.11031
  10000   0.06698   0.13979
 100000   0.71274   0.52627
```

```
1000000   7.02237   4.37249
Streams created with the iterate() method:
  Size    Sequential Parallel
   1000    0.08645   0.34696
  10000    0.35687   1.27861
 100000    3.24083  11.38709
1000000   29.92285 117.87909
Iterative solution with an explicit loop:
  Size    Iterative
   1000    0.00586
  10000    0.02330
 100000    0.22352
1000000    2.49677
```

Side Effects

Efficient execution of parallel streams that produces the desired results requires the stream operations (and their behavioral parameters) to avoid certain side effects.

- *Non-interfering behaviors*

  The behavioral parameters of stream operations should be non-interfering (p. 909)—both for sequential and parallel streams. Unless the stream data source is concurrent, the stream operations should not modify it during the execution of the stream. See building streams from collections (p. 897).

- *Stateless behaviors*

  The behavioral parameters of stream operations should be stateless (p. 909)— both for sequential and parallel streams. A behavioral parameter implemented as a lambda expression should not depend on any state that might change during the execution of the stream pipeline. The results from a stateful behavioral parameter can be nondeterministic or even incorrect. For a stateless behavioral parameter, the results are always the same.

  Shared state that is accessed by the behavior parameters of stream operations in a pipeline is not a good idea. Executing the pipeline in parallel can lead to *race conditions* in accessing the global state, and using synchronization code to provide thread-safety may defeat the purpose of parallelization. Using the three-argument `reduce()` or `collect()` method can be a better solution to encapsulate shared state.

  The intermediate operations `distinct()`, `skip()`, `limit()`, and `sorted()` are stateful (p. 915, p. 915, p. 917, p. 929).

See also Table 16.3, p. 938. They can carry extra performance overhead when executed in a parallel stream, as such an operation can entail multiple passes over the data and may require significant data buffering.

Ordering

An ordered stream (p. 891) processed by operations that preserve the encounter order will produce the same results, regardless of whether it is executed sequentially or in parallel. However, repeated execution of an unordered stream— sequential or parallel —can produce different results.

Preserving the encounter order of elements in an ordered parallel stream can incur a performance penalty. The performance of an ordered parallel stream can be improved if the ordering constraint is removed by calling the `unordered()` intermediate operation on the stream (p. 932).

The three stateful intermediate operations `distinct()`, `skip()`, and `limit()` can improve performance in a parallel stream that is unordered, as compared to one that is ordered (p. 915, p. 915, p. 917). The `distinct()` operation need only buffer *any* occurrence of a duplicate value in the case of an unordered parallel stream, rather than the *first* occurrence. The `skip()` operation can skip *any* `n` elements in the case of an unordered parallel stream, not necessarily the *first* `n` elements. The `limit()` operation can truncate the stream after *any* `n` elements in the case of an unordered parallel stream, and not necessarily after the *first* `n` elements.

The terminal operation `findAny()` is intentionally nondeterministic, and can return *any* element in the stream (p. 952). It is specially suited for parallel streams.

The `forEach()` terminal operation ignores the encounter order, but the `forEachOrdered()` terminal operation preserves the order (p. 948). The `sorted()` stateful intermediate operation, on the other hand, enforces a specific encounter order, regardless of whether it executed in a parallel pipeline (p. 929).

Autoboxing and Unboxing of Numeric Values

As the Stream API allows both object and numeric streams, and provides support for conversion between them (p. 934), choosing a numeric stream when possible can offset the overhead of autoboxing and unboxing in object streams.

As we have seen, in order to take full advantage of parallel execution, composition of a stream pipeline must follow certain rules to facilitate parallelization. In summary, the benefits of using parallel streams are best achieved when:

- The stream data source is of a sufficient size and the stream is easily splittable into substreams.
- The stream operations have no adverse side effects and are computation-intensive enough to warrant parallelization.

Review Questions

16.1 Given the following code:

Click here to view code image

```
import java.util.*;

public class RQ1 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("X", "XXX", "XX", "XXXX");
    int value = values.stream()
                      .mapToInt(v -> v.length())
                      .filter(v -> v != 4)
                      .reduce(1, (x, y) -> x * y);
    System.out.println(value);
  }
}
```

What is the result?

Select the one correct answer.

a. `4`
b. `6`
c. `24`
d. The program will throw an exception at runtime.

16.2 Which statement is true about the `Stream` methods?

a. The `filter()` method discards elements from the stream that match the given `Predicate`.
b. The `findFirst()` method always returns the first element in the stream.
c. The `reduce()` method removes elements from the stream that match the given `Predicate`.
d. The `sorted()` method sorts the elements in a stream

according to their natural order, or according to a given `Comparator`.

16.3 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ3 {
  public static void main(String[] args) {
    IntStream values = IntStream.range(0, 5);
    // (1) INSERT CODE HERE
    System.out.println(sum);
  }
}
```

Which of the following statements when inserted independently at (1) will result in a compile-time error? Select the two correct answers.

a. `int sum = values.reduce(0, (x, y) -> x + y);`

b. `int sum = values.parallel().reduce(0, (x, y) -> x + y);`

c. `int sum = values.reduce((x, y) -> x + y).orElse(0);`

d. `int sum = values.reduce(0, (x, y) -> x + y).orElse(0);`

e. `int sum = values.parallel().reduce((x, y) -> x + sum).orElse(0);`

f. `int sum = values.sum();`

16.4 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ4 {
  public static void main(String[] args) {
    IntStream values = IntStream.range(0, 5);
    // (1) INSERT CODE HERE
    System.out.println(value);
  }
}
```

Which of the following statements, when inserted independently at (1), will result in the value 4 being printed? Select the two correct answers.

a. `int value = values.reduce(0, (x, y) -> x + 1);`

b. `int value = values.reduce((x, y) -> x + 1).orElse(0);`

c. `int value = values.reduce(0, (x, y) -> y + 1);`

d. `int value = values.reduce(0, (x, y) -> y);`

e. `int value = values.reduce(1, (x, y) -> y + 1);`

f. `long value = values.count();`

16.5 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.stream.*;

public class RQ5 {
  public static void main(String[] args) {
    List<String> values = List.of("AA", "BBB", "C", "DD", "EEE");
    Map<Integer, List<String>> map = null;
    // (1) INSERT CODE HERE
    map.forEach((i, s) -> System.out.println(i + " " + s));
  }
}
```

Which statement when inserted independently at (1) will result in the output `1 [C]`?

Select the one correct answer.

a. Click here to view code image

```
map = values.stream()
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.filtering(s -> !s.contains("C"),
                             Collectors.toList())));
```

b. Click here to view code image

```
map = values.stream()
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.filtering(s -> s.contains("C"),
                             Collectors.toList())));
```

(c)

Click here to view code image

```
map = values.stream()
            .filter(s -> !s.contains("C"))
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.toList()));
```

c. Click here to view code image

```
map = values.stream()
            .filter(s -> s.contains("C"))
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.toList()));
```

16.6 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ7 {
  public static void main(String[] args) {
    Stream<String> values = Stream.generate(() -> "A");
    boolean value = values.peek(v -> System.out.print("B"))
                          .takeWhile(v -> !v.equals("A"))
                          .peek(v -> System.out.print("C"))
                          .anyMatch(v -> v.equals("A"));
    System.out.println(value);
  }
}
```

What is the result?

Select the one correct answer.

a. `Btrue`
b. `Ctrue`
c. `BCtrue`
d. `Bfalse`
e. `Cfalse`
f. `BCfalse`

16.7 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ9 {
  public static void main(String[] args) {
    IntStream.range('a', 'e')
             .mapToObj(i -> String.valueOf((char) i).toUpperCase())
             .filter(s -> "AEIOU".contains(s))
             .forEach(s -> System.out.print(s));
  }
}
```

What is the result?

Select the one correct answer.

a. `A`
b. `AE`
c. `BCD`
d. The program will fail to compile.

16.8 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ10 {
  public static void main(String[] args) {
    IntStream.range(0, 5)
             .filter(i -> i % 2 != 0)
             .forEach(i -> System.out.println(i));
  }
}
```

Which of the following statements will produce the same result as the program? Select the two correct answers.

a. Click here to view code image

```
IntStream.rangeClosed(0, 5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

b. Click here to view code image

```
IntStream.range(0, 10)
```

```
         .takeWhile(i -> i < 5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

c. Click here to view code image

```
IntStream.range(0, 10)
         .limit(5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

d. Click here to view code image

```
IntStream.generate(() -> {int x = 0; return x++;})
         .takeWhile(i -> i < 4)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

e. Click here to view code image

```
var x = 0;
IntStream.generate(() -> return x++)
         .limit(5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

16.9 Given the following code:

Click here to view code image

```
import java.util.function.*;
import java.util.stream.*;

public class RQ11 {
  public static void main(String[] args) {
    Stream<String> abc = Stream.of("A", "B", "C");
    Stream<String> xyz = Stream.of("X", "Y", "Z");
    String value = Stream.concat(xyz, abc).reduce((a, b) -> b +
    System.out.println(value);
  }
}
```

What is the result?

Select the one correct answer.

a. `ABCXYZ`

b. `XYZABC`

(c) `ZYXCBA`

f. `CBAZYX`

16.10 Which statement produces a different result from the other statements?

Select the one correct answer.

a. Click here to view code image

```
Stream.of("A", "B", "C", "D", "E")
      .filter(s -> s.compareTo("B") < 0)
      .collect(Collectors.groupingBy(s -> "AEIOU".contains(s)))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

b. Click here to view code image

```
Stream.of("A", "B", "C", "D", "E")
      .filter(s -> s.compareTo("B") < 0)
      .collect(Collectors.partitioningBy(s -> "AEIOU".contains(s)))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

c. Click here to view code image

```
Stream.of("A", "B", "C", "D", "E")
      .collect(Collectors.groupingBy(s -> "AEIOU".contains(s),
                   Collectors.filtering(s -> s.compareTo("B") < 0,
                                        Collectors.toList())))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

d. Click here to view code image

```
Stream.of("A", "B", "C", "D", "E")
      .collect(Collectors.partitioningBy(s -> "AEIOU".contains(s),
                   Collectors.filtering(s -> s.compareTo("B") < 0,
                                        Collectors.toList())))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

16.11 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ13 {
  public static void main(String[] args) {
    Stream<String> strings = Stream.of("i", "am", "ok").paralle
    IntStream chars = strings.flatMapToInt(line -> line.chars()
    chars.forEach(c -> System.out.print((char)c));
  }
}
```

What is the result?
Select the one correct answer.

a. `iamok`
b. `aikmo`
c. `amiok`
d. The result from running the program is unpredictable.
e. The program will throw an exception at runtime.

16.12 Which of the following statements are true about the `Stream` methods? Select the two correct answers.

a. The `filter()` method accepts a `Function`.
b. The `peek()` method accepts a `Function`.
c. The `peek()` method accepts a `Consumer`.

(d) The `forEach()` method accepts a `Consumer`.
e. The `map()` method accepts a `Predicate`.
f. The `max()` method accepts a `Predicate`.

g. The `findAny()` method accepts a `Predicate`.

16.13 Which `Stream` methods are terminal operations? Select the two correct answers.

a. `peek()`
b. `forEach()`
c. `map()`
d. `filter()`
e. `sorted()`
f. `min()`

16.14 Which `Stream` methods have short-circuit execution? Select the two correct answers.

a. `collect()`
b. `limit()`
c. `flatMap()`
d. `anyMatch()`
e. `reduce()`
f. `sum()`

16.15 Given the following code:

Click here to view code image

```
import java.util.stream.*;

public class RQ17 {
  public static void main(String[] args) {
    Stream<String> values = Stream.of("is", "this", "", null, "ok", "?")
    // (1) INSERT CODE HERE
    System.out.println(c);
  }
}
```

Which statement inserted independently at (1) produces the output `6`?
Select the one correct answer.

a. `long c = values.count();`
b. `long c = values.collect(Collectors.counting());`
c. `int c = values.mapToInt(v -> 1).reduce(0, (x, y) -> x + 1);`
d. `long c = values.collect(Collectors.reducing(0L, v -> 1L, Long::sum));`
e. `int c = values.mapToInt(v -> 1).sum();`
f. Insert any of the above.

16.16 Which code produces identical results? Select the two correct answers.

a. Click here to view code image

```
Set<String> set1 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                         .filter(v -> v != null)
                         .collect(Collectors.toSet());
set1.stream()
    .mapToInt(v -> v.length())
    .sorted()
    .forEach(v -> System.out.print(v));
```

b. Click here to view code image

```
Set<Integer> set2 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                          .map(v -> (v == null) ? 0 : v.length())
                          .filter(v -> v != 0)
                          .collect(Collectors.toSet());
set2.stream()
    .sorted()
    .forEach(v -> System.out.print(v));
```

c. Click here to view code image

```
List<Integer> list1 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                            .map(v -> (v == null) ? 0 : v.length())
                            .filter(v -> v != 0)
                            .toList();
list1.stream()
     .sorted()
     .forEach(v -> System.out.print(v));
```

d. Click here to view code image

```
List<Integer> list2 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                            .map(v -> (v == null) ? 0 : v.length())
                            .filter(v -> v != 0)
                            .distinct()
                            .toList();
 list2.stream()
      .sorted()
      .forEach(v -> System.out.print(v));
```

Date and Time 17

Chapter Topics

- Overview of the new Date and Time API in the `java.time` package
- Understanding the temporal concepts represented by the `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, `Instant`, `Period`, and `Duration` classes
- Creating and using temporal objects
- Accessing temporal objects using temporal units and temporal fields
- Comparing temporal objects
- Creating modified instances of temporal objects
- Performing temporal arithmetic with temporal objects
- Using time zones and daylight savings with `ZonedDateTime` objects.
- Interoperability between date/time values and legacy dates

Java SE 17 Developer Exam O	
[1.3] Manipulate date, time, duration, period, instant and time-zone objects using Date-Time API	*§17.1, p. 1024* *to* *§17.7, p. 1072*

Java 8 introduced a new and comprehensive API for date and time. This chapter provides comprehensive coverage of essential topics regarding the Date and Time API. Coverage of formatting and parsing of date and time values is deferred to §18.6, p. 1127.

17.1 Date and Time API Overview

The `java.time` package provides the main support for dealing with dates and times. It contains the main classes that represent date and time values, including those that represent an amount of time.

- `LocalDate`: This class represents a date in terms of *date fields* (year, month, day). A *date* has no time fields or a time zone. (This class is not to be confused with the `java.util.Date` legacy class.)
- `LocalTime`: This class represents time in a 24-hour day in terms of *time fields* (hour, minute, second, nanosecond). A *time* has no date fields or a time zone.
- `LocalDateTime`: This class represents the concept of date and time combined, in terms of *both* date and time fields. A *date-time* has no time zone.
- `ZonedDateTime`: This class represents the concept of a date-time with a time zone— that is, a *zoned date-time*.
- `Instant`: This class represents a measurement of time as a point on a timeline starting from a specific origin (called the *epoch*). An *instant* is represented with nanosecond precision and can be a negative value.
- `Period`: This class represents an amount or quantity of time in terms of number of days, months, and years, which can be negative. A *period* is a *date-based* amount of time. It has no notion of a clock time, a date, or a time zone.
- `Duration`: This class represents an amount or quantity of time in terms of number of seconds and nanoseconds, which can be negative. A *duration* is a *time-based* amount of time. As with instants, durations have no notion of a clock time, a date, or a time zone.

We will use the term *temporal objects* to mean objects of classes that represent temporal concepts.

The temporal classes implement *immutable* and *thread-safe* temporal objects. The state of an immutable object cannot be changed. Any method that is supposed to modify such an object returns a modified copy of the temporal object. It is a common mistake to ignore the object returned, thinking that the current object has been modified. Thread-safety guarantees that the state of such an object is not compromised by concurrent access.

Table 17.1 summarizes the fields in selected classes from the Date and Time API. The table shows the relative size of the objects of these classes in terms of their fields; for example, a `LocalTime` has only time fields, whereas a `ZonedDateTime` has time-, date-, and zone-based fields. The three asterisks `***` indicate that this information can be derived by methods provided by the class, even though these fields do not exist in an object of the `Duration` class.

Table 17.1 *Fields in Selected Classes in the Date and Time API*

[illegible]								
`LocalTime` (p. 1027)				▨	▨	▨		
`Local` (p. 1027)	▨	▨	▨					
`Local` (p. 1027)	▨	▨	▨	▨	▨	▨		
`Zoned` (p. 1072)	▨	▨	▨	▨	▨	▨	▨	▨
`Instant` (p. 1049)						▨		
`Perio` (p. 1057)	▨	▨	▨					
*** `Duration` (p. 1064)						▨		

The information in Table 17.1 is crucial to understanding how the objects of these classes can be used. A common mistake is to access, format, or parse a temporal object that does not have the required temporal fields. For example, a `LocalTime` object has only time fields, so trying to format it with a formatter for date fields will result in a `java.time.DateTimeException`. Many methods will also throw an exception if an invalid or an out-of-range argument is passed in the method call. It is important to keep in mind which temporal fields constitute the state of a temporal object.

Table 17.2 provides an overview of the method naming conventions used in the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, and `Instant`. This method naming convention makes it easy to use the API, as it ensures method naming is standardized across all temporal classes. Depending on the method, the suffix *XXX* in a method name can be a specific field (e.g., designate the `Year` field in the `getYear` method name), a specific unit (e.g., designate the unit `Days` for number of days in the `plusDays` method name), or a class name (e.g., designate the class type in the `toLocalDate` method name).

Table 17.2 *Selected Method Name Prefixes in the Temporal Classes*

Prefix (parameters not shown)	Usage
`at`*`XXX`*`()`	Create a new temporal object by combining this temporal object with another temporal object. Not provided by the `ZonedDateTime` class.
`of()` `of`*`XXX`*`()`	Static factory methods for constructing temporal objects from constituent temporal fields.
`get()` `get`*`XXX`*`()`	Access specific fields in this temporal object.
`is`*`XXX`*`()`	Check specific properties of this temporal object.
`minus()` `minus`*`XXX`*`()`	Return a copy of this temporal object after subtracting an amount of time.
`plus()` `plus`*`XXX`*`()`	Return a copy of this temporal object after adding an amount of time.
`to`*`XXX`*`()`	Convert this temporal object to another type.
`with()` `with`*`XXX`*`()`	Create a copy of this temporal object with one field modified.

Apart from the methods shown in Table 17.2, the selected methods shown in Table 17.3 are common to the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, `ZonedDateTime`, and `Instant`.

Table 17.3 *Selected Common Methods in the Temporal Classes*

Method (parameters not shown	Usage
`now()`	Static method that obtains the current time from the system or specified clock in the default or specified time zone.
`from()`	Static method to obtain an instance of this temporal class from another temporal.
`until()`	Calculate the amount of time from this temporal object to another temporal object.
`toString()`	Create a text representation of this temporal object.
`equals()`	Compare two temporal objects for equality.
`hashCode()`	Returns a hash code for this temporal object.
`compareTo()`	Compare two temporal objects. (The class `ZonedDateTime` does not implement the `Comparable<E>` interface.)
`parse()`	Static method to obtain a temporal instance from a specified text string (§18.6, p. 1127).
`format()`	Create a text representation of this temporal object using a

specified formatter (§18.6, p. 1127).
(`Instant` class does not provide this method.)

Subsequent sections in this chapter provide ample examples of how to create, combine, convert, access, and compare temporal objects, including the use of temporal arithmetic and dealing with time zones and daylight savings. For formatting and parsing temporal objects, see §18.6, p. 1127.

17.2 Working with Dates and Times

The classes `LocalTime`, `LocalDate`, and `LocalDateTime` in the `java.time` package represent time-based, date-based, and combined date-based and time-based temporal objects, respectively, that are all time zone agnostic. These classes represent *human time* that is calendar-based, meaning it is defined in terms of concepts like year, month, day, hour, minute, and second, that humans use. The `Instant` class can be used to represent *machine time*, which is defined as a point measured with nanosecond precision on a continuous timeline starting from a specific origin (p. 1049).

Time zones and daylight savings are discussed in §17.7, p. 1072.

Creating Dates and Times

The temporal classes in the `java.time` package do not provide any `public` constructors to create temporal objects. Instead, they provide overloaded static factory methods named `of` which create temporal objects from constituent temporal fields. We use the term *temporal fields* to mean both time fields (hours, minutes, seconds, nanoseconds) and date fields (year, month, day). The `of()` methods check that the values of the arguments are in range. Any invalid argument results in a `java.time.DateTimeException`.

Click here to view code image

```
// LocalTime
static LocalTime of(int hour, int minute)
static LocalTime of(int hour, int minute, int second)
static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
static LocalTime ofSecondOfDay(long secondOfDay)
```

These static factory methods in the `LocalTime` class return an instance of `Local-Time` based on the specified values for the specified time fields. The second and nanosecond fields are set to zero, if not specified.

The last method accepts a value for the `secondOfDay` parameter in the range [0, 24 * 60 * 60 – 1] to create a `LocalTime`.

Click here to view code image

```
// LocalDate
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
static LocalDate ofYearDay(int year, int dayOfYear)
```

These static factory methods in the `LocalDate` class return an instance of `LocalDate` based on the specified values for the date fields. The `java.time.Month` enum type allows months to be referred to by name—for example, `Month.MARCH`. Note that month numbering starts with 1 (`Month.JANUARY`).

The last method creates a date from the specified year and the day of the year.

Click here to view code image

```
// LocalDateTime
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
                        int minute, int second, int nanoOfSecond)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second, int nanoOfSecond)
```

These static factory methods in the `LocalDateTime` class return an instance of `LocalDateTime` based on the specified values for the time and date fields. The second and nanosecond fields are set to zero, if not specified. The `java.time.Month` enum type allows months to be referred to by name—for example, `Month.MARCH` (i.e., month 3 in the year).

Click here to view code image

```
static LocalDateTime of(LocalDate date, LocalTime time)
```

Combines a `LocalDate` and a `LocalTime` into a `LocalDateTime`.

All code snippets in this subsection can be found in Example 17.1, p. 1031, ready for running and experimenting. An appropriate `import` statement with the `java.time` package should be included in the source file to access any of the temporal classes by their simple name.

The `LocalTime` *Class*

The declaration statements below show examples of creating instances of the `LocalTime` class to represent time on a 24-hour clock in terms of hours, minutes, seconds, and nanoseconds.

Click here to view code image

```
LocalTime time1 = LocalTime.of(8, 15, 35, 900);   // 08:15:35.000000900
LocalTime time2 = LocalTime.of(16, 45);           // 16:45
// LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
```

The ranges of values for time fields hour (0–23), minute (0–59), second (0–59), and nanosecond (0–999,999,999) are defined by the ISO standard. The `toString()` method of the class will format the time fields according to the ISO standard as follows:

```
HH:mm:ss.SSSSSSSSS
```

Omitting the seconds (`ss`) and fractions of seconds (`SSSSSSSSS`) in the call to the `of()` method implies that their value is zero. (More on formatting in §18.6, p. 1134.) In the second declaration statement above, the seconds and the nanoseconds are not specified in the method call, resulting in their values being set to zero. In the third statement, the value of the hour field (25) is out of range, and if the statement is uncommented, it will result in a `DateTimeException`.

The `LocalDate` *Class*

Creating instances of the `LocalDate` class is analogous to creating instances of the `LocalTime` class. The `of()` method of the `LocalDate` class is passed values for date fields: the year, month of the year, and day of the month.

Click here to view code image

```
LocalDate date1 = LocalDate.of(1969, 7, 20);              // 1969-07-20
LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);        // DateTimeException
//  LocalDate date4 = LocalDate.of(2021, 2, 29);         // DateTimeException
```

The ranges of the values for date fields year, month, and day are (–999,999,999 to +999,999,999), (1–12), and (1–31), respectively. The month can also be specified using the enum constants of the `java.time.Month` class, as in the second declaration statement above. A `DateTimeException` is thrown if the value of any parameter is out of range, or if the day is invalid for the specified month of the year. In the third declaration, the value of the month field 13 is out of range. In the last declaration, the month of February cannot have 29 days, since the year 2021 is not a leap year.

The `toString()` method of the `LocalDate` class will format the date fields according to the ISO standard (§18.6, p. 1134):

```
uuuu-MM-dd
```

The year is represented as a *proleptic year* in the ISO standard, which can be negative. A year in CE (Current Era, or AD) has the same value as a proleptic year; for example, 2021 CE is the same as the proleptic year 2021. However, for a year in BCE (Before Current Era, or BC), the proleptic year 0 corresponds to 1 BCE, the proleptic year –1 corresponds to 2 BCE, and so on. In the second declaration in the preceding set of examples, the date `-3113-08-11` corresponds to 11 August 3114 BCE.

The `LocalDateTime` *Class*

The class `LocalDateTime` allows the date and the time to be combined into one entity, which is useful for representing such concepts as appointments that require both a time and a date. The `of()` methods in the `LocalDateTime` class are combinations of the `of()` methods from the `LocalTime` and `LocalDate` classes, taking values of both time and date fields as arguments. The `toString()` method of this class will format the temporal fields according to the ISO standard (§18.6, p. 1134):

```
uuuu-MM-ddTHH:mm:ss.SSSSSSSSS
```

The letter `T` separates the values of the date fields from those of the time fields.

Click here to view code image

```
// 2021-04-28T12:15
LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
// 2021-08-19T14:00
LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 19, 14, 0);
```

The `LocalDateTime` class also provides an `of()` method that combines a `LocalDate` object and a `LocalTime` object. The first declaration in the next code snippet combines a date and a time. The static field `LocalTime.NOON` defines the time at noon. In addition, the `LocalTime` class provides the instance method `atDate()`, which takes a date as an argument and returns a `LocalDateTime` object. The second declaration combines the time at noon with the date referred to by the reference `date1`. Conversely, the `LocalDate` class provides the overloaded instance method `atTime()` to combine a date with a specified time. In the last two declarations, the `atTime()` method is passed a `LocalTime` object and values for specific time fields, respectively.

Click here to view code image

```
// LocalDate date1 is 1969-07-20.
LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON); // 1969-07-20T12:
LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);       // 1969-07-20T12:
LocalDateTime dt5 = date1.atTime(LocalTime.NOON);            // 1969-07-20T12:
LocalDateTime dt6 = date1.atTime(12, 0);                     // 1969-07-20T12:
```

As a convenience, each temporal class provides a static method `now()` that reads the system clock and returns the values for the relevant temporal fields in an instance of the target class.

Click here to view code image

```
LocalTime currentTime = LocalTime.now();
LocalDate currentDate = LocalDate.now();
LocalDateTime currentDateTime = LocalDateTime.now();
```

Example 17.1 includes the different ways to create temporal objects that we have discussed so far.

Click here to view code image

```
// LocalTime
LocalDateTime atDate(LocalDate date)
```

Returns a `LocalDateTime` that combines this time with the specified date.

Click here to view code image

```
// LocalDate
LocalDateTime atTime(LocalTime time)
LocalDateTime atTime(int hour, int minute)
LocalDateTime atTime(int hour, int minute, int second)
LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond)
LocalDateTime atStartOfDay()
```

Return a `LocalDateTime` that combines this date with the specified values for time fields. The second and nanosecond fields are set to zero, if their values are not specified. In the last method, this date is combined with the time at midnight.

Click here to view code image

```
// LocalDateTime
```

```
ZonedDateTime atZone(ZoneId zone)
```

Returns a `ZonedDateTime` by combining this date-time with the specified time zone (p. 1072).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime, respectively.
static LocalTime now()
static LocalDate now()
static LocalDateTime now()
```

Each temporal class has this static factory method, which returns either the current time, date, or date-time from the system clock.

Example 17.1 *Creating Local Dates and Local Times*

Click here to view code image

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;

public class CreatingTemporals {

  public static void main(String[] args) {

    // Creating a specific time from time-based values:
    LocalTime time1 = LocalTime.of(8, 15, 35, 900);// 08:15:35.000000900
    LocalTime time2 = LocalTime.of(16, 45);        // 16:45
//  LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
    System.out.println("Surveillance start time: " + time1);
    System.out.println("Closing time: " + time2);

    // Creating a specific date from date-based values:
    LocalDate date1 = LocalDate.of(1969, 7, 20);            // 1969-07-20
    LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);           // DateTimeExcepti
//  LocalDate date4 = LocalDate.of(2021, 2, 29);            // DateTimeExcepti
    System.out.println("Date of lunar landing:        " + date1);
    System.out.println("Start Date of Mayan Calendar: " + date2);

    // Creating a specific date-time from date- and time-based values.
    // 2021-04-28T12:15
    LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
    // 2021-08-17T14:00
    LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 17, 14, 0);
    System.out.println("Car service appointment: " + dt1);
    System.out.println("Hospital appointment:    " + dt2);

    // Combining date and time objects.
    // 1969-07-20T12:00
    LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON);
    LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);
    LocalDateTime dt5 = date1.atTime(LocalTime.NOON);
    LocalDateTime dt6 = date1.atTime(12, 0);
    System.out.println("Factory date-time combo: " + dt3);
    System.out.println("Time with date combo:     " + dt4);
    System.out.println("Date with time combo:     " + dt5);
    System.out.println("Date with explicit time combo: " + dt6);

    // Current time:
    LocalTime currentTime = LocalTime.now();
    System.out.println("Current time:      " + currentTime);

    // Current date:
    LocalDate currentDate = LocalDate.now();
    System.out.println("Current date:      " + currentDate);

    // Current date and time:
    LocalDateTime currentDateTime = LocalDateTime.now();
    System.out.println("Current date-time: " + currentDateTime);
  }
}
```

Possible output from the program:

Click here to view code image

```
Surveillance start time: 08:15:35.000000900
Closing time: 16:45
Date of lunar landing:         1969-07-20
Start Date of Mayan Calendar: -3113-08-11
Car service appointment: 2021-04-28T12:15
Hospital appointment:    2021-08-17T14:00
Factory date-time combo: 1969-07-20T12:00
Time with date combo:    1969-07-20T12:00
Date with time combo:    1969-07-20T12:00
Date with explicit time combo: 1969-07-20T12:00
Current time:      10:55:41.296744
Current date:      2021-03-05
Current date-time: 2021-03-05T10:55:41.299318
```

Accessing Fields in Dates and Times

A temporal object provides get methods that are tailored to access the values of specific temporal fields that constitute its state. The `LocalTime` and `LocalDate` classes provide get methods to access the values of time and date fields, respectively. Not surprisingly, the `LocalDateTime` class provides get methods for accessing the values of both time and date fields.

> Click here to view code image
>
> ```
> // LocalTime, LocalDateTime
> int getHour()
> int getMinute()
> int getSecond()
> int getNano()
> ```
>
> Return the value of the appropriate time field from the current `LocalTime` or `LocalDateTime` object.

```
// LocalDate, LocalDateTime
int       getDayOfMonth()
DayOfWeek getDayOfWeek()
int       getDayOfYear()
Month     getMonth()
int       getMonthValue()
int       getYear()
```

Return the value of the appropriate date field from the current `LocalDate` or `LocalDateTime` object. The enum type `DayOfWeek` allows days of the week to be referred to by name; for example, `DayOfWeek.MONDAY` is day 1 of the week. The enum type `Month` allows months of the year to be referred to by name—for example, `Month.JANUARY`. The month value is from 1 (`Month.JANUARY`) to 12 (`Month.DECEMBER`).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
int get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

The first two methods return the value of the specified `TemporalField` (p. 1046) from this temporal object as an `int` value or as a `long` value, respectively. To specify fields whose value does not fit into an `int`, the `getLong()` method must be used.

The third method checks if the specified field is supported by this temporal object. It avoids an exception being thrown if it has been determined that the field is supported.

Using an invalid field in a get method will result in any one of these exceptions: `DateTimeException` (field value cannot be obtained), `UnsupportedTemporalType-Exception` (field is not supported), or `ArithmeticException` (numeric overflow occurred).

Here are some examples of using the get methods; more examples can be found in Example 17.2. Given that `time` and `date` refer to a `LocalTime` (`08:15`) and a `LocalDate` (`1945-08-06`), respectively, the code below shows how we access the values of the temporal fields using specifically named get methods and using specific temporal fields.

Click here to view code image

```
int minuteOfHour1 = time.getMinute();                     // 15
int minuteOfHour2 = time.get(ChronoField.MINUTE_OF_HOUR); // 15

int monthVal1 = date.getMonthValue();                     // 8
int monthVal2 = date.get(ChronoField.MONTH_OF_YEAR);      // 8
```

The temporal class `LocalDateTime` also provides two methods to obtain the date and the time as temporal objects, in contrast to accessing the values of individual date and time fields.

Click here to view code image

```
LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
LocalDate date = doomsday.toLocalDate();                  // 1945-08-06
LocalTime time = doomsday.toLocalTime();                  // 08:15
```

```
// LocalDateTime
LocalDate toLocalDate()
LocalTime toLocalTime()
```

These methods can be used to get the `LocalDate` and the `LocalTime` components of this date-time object, respectively.

The following two methods return the number of days in the month and in the year represented by a `LocalDate` object.

Click here to view code image

```
LocalDate foolsday = LocalDate.of(2022, 4, 1);
int daysInMonth = foolsday.lengthOfMonth();    // 30
int daysInYear = foolsday.lengthOfYear();     // 365 (2022 is not a leap year.
```

```
// LocalDate
int lengthOfMonth()
int lengthOfYear()
```

These two methods return the number of days in the month and in the year represented by this date, respectively.

Comparing Dates and Times

It is also possible to check whether a temporal object represents a point in time before or after another temporal object of the same type. In addition, the `LocalDate` and `LocalDateTime` classes provide an `isEqual()` method that determines whether a temporal object is equal to another temporal object of the *same* type. In contrast, the `equals()` method allows equality comparison with an *arbitrary* object.

Click here to view code image

```
LocalDate d1 = LocalDate.of(1948, 2, 28);                  // 1948-02-28
LocalDate d2 = LocalDate.of(1949, 3, 1);                   // 1949-03-01
boolean result1 = d1.isBefore(d2);                         // true
boolean result2 = d2.isAfter(d1);                          // true
boolean result3 = d1.isAfter(d1);                                // false
boolean result4 = d1.isEqual(d2);                                // false
boolean result5 = d1.isEqual(d1);                                // true
boolean result6 = d1.isLeapYear();                               // true
```

The temporal classes implement the `Comparable<E>` interface, providing the `compareTo()` method so that temporal objects can be compared in a meaningful way. The temporal classes also override the `equals()` and the `hashCode()` methods of the `Object` class. These methods make it possible to both search for and sort temporal objects.

Click here to view code image

```
// LocalTime
boolean isBefore(LocalTime other)
boolean isAfter(LocalTime other)
```

Determine whether this `LocalTime` represents a point on the timeline before or after the `other` time, respectively.

Click here to view code image

```
// LocalDate
boolean isBefore(ChronoLocalDate other)
boolean isAfter(ChronoLocalDate other)
boolean isEqual(ChronoLocalDate other)
boolean isLeapYear()
```

The first two methods determine whether this `LocalDate` represents a point on the timeline before or after the `other` date, respectively. The `LocalDate` class implements the `ChronoLocalDate` interface.

The third method determines whether this date is equal to the specified date.

The last method checks for a leap year according to the ISO proleptic calendar system rules.

Click here to view code image

```
// LocalDateTime
boolean isBefore(ChronoLocalDateTime<?> other)
boolean isAfter(ChronoLocalDateTime<?> other)
boolean isEqual(ChronoLocalDateTime<?> other)
```

The first two methods determine whether this `LocalDateTime` represents a point on the timeline before or after the specified date-time, respectively. The `Local-DateTime` class implements the `ChronoLocalDateTime<LocalDateTime>` interface.

The third method determines whether this date-time object

> represents the same point on the timeline as the other date-time.
>
> Click here to view code image
>
> ```
> int compareTo(LocalTime other)                  // LocalTime
> int compareTo(ChronoLocalDate other)            // LocalDate
> int compareTo(ChronoLocalDateTime<?> other)     // LocalDateTime
> ```
>
> Compare this temporal object to another temporal object. The three temporal classes implement the `Comparable<E>` functional interface. The `compareTo()` method returns `0` if the two temporal objects are equal, a negative value if this temporal object is less than the other temporal object, and a positive value if this temporal object is greater than the other temporal object.

Creating Modified Copies of Dates and Times

An immutable object does not provide any set methods that can change its state. Instead, it usually provides what are known as `with` methods (or *withers*) that return a copy of the original object where exactly one field has been set to a new value. The `LocalTime` and `LocalDate` classes provide `with` methods to set the value of a time or date field, respectively. Not surprisingly, the `LocalDateTime` class provides `with` methods to set the values of both time and date fields individually. A `with` method changes a specific property in an absolute way, which is reflected in the state of the new temporal object; the original object, however, is not affected. Such `with` methods are also called *absolute adjusters*, in contrast to the *relative adjusters* that we will meet later (p. 1040).

> Click here to view code image
>
> ```
> // LocalTime, LocalDateTime
> LocalTime/LocalDateTime withHour(int hour)
> LocalTime/LocalDateTime withMinute(int minute)
> LocalTime/LocalDateTime withSecond(int second)
> LocalTime/LocalDateTime withNano(int nanoOfSecond)
> ```
>
> Return a copy of this `LocalTime` or `LocalDateTime` with the value of the appropriate time field changed to the specified value. A `DateTimeException` is thrown if the argument value is out of range.
>
> Click here to view code image
>
> ```
> // LocalDate, LocalDateTime
> LocalDate/LocalDateTime withYear(int year)
> LocalDate/LocalDateTime withMonth(int month)
> ```

```
LocalDate/LocalDateTime withDayOfMonth(int dayOfMonth)
LocalDate/LocalDateTime withDayOfYear(int dayOfYear)
```

Return a copy of this `LocalDate` or `LocalDateTime` with the value of the appropriate date field changed to the specified value. A `DateTimeException` is thrown if the specified value is out of range or is invalid in combination with the values of the other time or date fields in the temporal object.

The first and second methods will adjust the day of the month to the *last valid day* of the month, if the day of the month becomes invalid when the year or the month is changed (e.g., the month value 2 will change the date 2020-03-31 to 2020-02-29).

In contrast, the third method will throw a `DateTimeException` if the specified day of the month is invalid for the month-year combination (e.g., the day of month 29 is invalid for February 2021), as will the last method if the day of the year is invalid for the year (e.g., the day of year 366 is invalid for the year 2021).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime with(TemporalField field, long newValue)
```

Returns a copy of this temporal object with the specified `TemporalField` (p. 1046) set to a new value. The `ChronoField` enum type implements the `TemporalField` interface, and its enum constants define specific temporal fields (p. 1046).

Using an invalid field in the `with()` method will result in any one of these exceptions: `DateTimeException` (field value cannot be set), `Unsupported-TemporalTypeException` (field is not supported), or `ArithmeticException` (numeric overflow occurred).

The code lines below are from Example 17.2. In the second assignment statement, the method calls are chained. Three instances of the `LocalDate` class are created consecutively, as each `with` method is called to set the value of a specific date field. The last assignment shows the use of temporal fields in the `with()` method for the same purpose.

Click here to view code image

```
LocalDate date2 = LocalDate.of(2021, 3, 1);                      // 2021-03-01
date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28); // 2024-02-28
```

```
LocalDate date3 = LocalDate.of(2021, 3, 1);                       // 2021-03-01
date3 = date3
    .with(ChronoField.YEAR, 2024L)
    .with(ChronoField.MONTH_OF_YEAR, 2L)
    .with(ChronoField.DAY_OF_MONTH, 28L);                         // 2024-02-28
```

The following code contains a logical error, such that the last two `LocalDate` instances returned by the `with` methods are ignored, and the reference `date2` never gets updated.

Click here to view code image

```
date2 = date2.withYear(2022);                          // 2022-03-01
date2.withMonth(2).withDayOfMonth(28);                 // date2 is still 2022-03-0
```

In the next code examples, each call to a `with` method throws a `DateTimeException`. The minute and hour values are out of range for a `LocalTime` object. Certainly the month value 13 is out of range for a `LocalDate` object. The day of month value 31 is not valid for April, which has 30 days. The day of year value 366 is out of range as well, since the year 2021 is not a leap year.

Click here to view code image

```
LocalTime time = LocalTime.of(14, 45);       // 14:45
time = time.withMinute(100);       // Out of range. DateTimeException.
time = time.withHour(25);          // Out of range. DateTimeException.

LocalDate date = LocalDate.of(2021, 4, 30);  // 2021-04-30
date = date.withMonth(13);         // Out of range. DateTimeException.
date = date.withDayOfMonth(31);    // Out of range for month. DateTimeExceptio
date = date.withDayOfYear(366);    // Out of range for year. DateTimeException
```

The code snippets below illustrate how the `withYear()` and `withMonth()` methods adjust the day of the month, if necessary, when the year or the month is changed, respectively.

Click here to view code image

```
LocalDate date3 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date3 = date3.withYear(2021);                 // Expected: 2021-02-29
System.out.println("Date3: " + date3);        // Adjusted: 2021-02-28

LocalDate date4 = LocalDate.of(2021, 3, 31);  // Original: 2021-03-31
date4 = date4.withMonth(4);                   // Expected: 2021-04-31
System.out.println("Date4: " + date4);        // Adjusted: 2021-04-30
```

The year in the date 2020-02-29 is changed to 2021, resulting in the following date: 2021-02-29. Since the year 2021 is not a leap year, the month of February cannot have 29 days. The `withYear()` method adjusts the day of the month to the last valid day of the month (i.e., 28). Similarly, the month in the date 2021-03-31 is changed to 4 (i.e., April), resulting in the following date:

2021-04-31. Since the month April has 30 days, the `withMonth()` method adjusts the day of the month to the last valid day of the month (i.e., 30).

Example 17.2 *Using Local Dates and Local Times*

Click here to view code image

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.temporal.ChronoField;

public class UsingTemporals {

  public static void main(String[] args) {
    // Date-Time: 1945-08-06T08:15
    LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
    LocalDate date = doomsday.toLocalDate();                   // 1945-08-06
    LocalTime time = doomsday.toLocalTime();                   // 08:15
    System.out.println("Date-Time: " + doomsday);
    System.out.println();

    // Time: 08:15
    int hourOfDay      = time.getHour();                       // 8
    int minuteOfHour1  = time.getMinute();                     // 15
    int minuteOfHour2  = time.get(ChronoField.MINUTE_OF_HOUR); // 15
    int secondOfMinute = time.getSecond();                     // 0
    System.out.println("Time of day:      " + time);
    System.out.println("Hour-of-day:      " + hourOfDay);
    System.out.println("Minute-of-hour 1: " + minuteOfHour1);
    System.out.println("Minute-of-hour 2: " + minuteOfHour2);
    System.out.println("Second-of-minute: " + secondOfMinute);
    System.out.println();

    // Date: 1945-08-06
    int year       = date.getYear();                           // 1945
    int monthVal1  = date.getMonthValue();                     // 8
    int monthVal2  = date.get(ChronoField.MONTH_OF_YEAR);      // 8
    Month month    = date.getMonth();                          // AUGUST
    DayOfWeek dow  = date.getDayOfWeek();                      // MONDAY
    int day        = date.getDayOfMonth();                     // 6
    System.out.println("Date:  " + date);
    System.out.println("Year:  " + year);
    System.out.println("Month value 1: " + monthVal1);
    System.out.println("Month value 2: " + monthVal2);
    System.out.println("Month-of-year: " + month);
    System.out.println("Day-of-week:   " + dow);
    System.out.println("Day-of-month:  " + day);
    System.out.println();

    // Ordering
    LocalDate d1 = LocalDate.of(1948, 2, 28);                  // 1948-02-28
    LocalDate d2 = LocalDate.of(1949, 3, 1);                   // 1949-03-01
    boolean result1 = d1.isBefore(d2);                         // true
    boolean result2 = d2.isAfter(d1);                          // true
    boolean result3 = d1.isAfter(d1);                          // false
    boolean result4 = d1.isEqual(d2);                          // false
    boolean result5 = d1.isEqual(d1);                         // true
    boolean result6 = d1.isLeapYear();                        // true

    System.out.println("Ordering:");
    System.out.println(d1 + " is before "   + d2 + ": " + result1);
    System.out.println(d2 + " is after "    + d1 + ": " + result2);
    System.out.println(d1 + " is after "    + d1 + ": " + result3);
    System.out.println(d1 + " is equal to " + d2 + ": " + result4);
    System.out.println(d1 + " is equal to " + d1 + ": " + result5);
    System.out.println(d1.getYear() + " is a leap year: " + result6);
    System.out.println();

    System.out.println("Using absolute adjusters:");
    LocalDate date2 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date2);    // 2021-03-01
    date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28);
    System.out.println("Date after adjusting:  " + date2);    // 2024-02-28
    System.out.println();

    System.out.println("Using temporal fields:");
    LocalDate date3 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date3);    // 2021-03-01
    date3 = date3
        .with(ChronoField.YEAR, 2024L)
        .with(ChronoField.MONTH_OF_YEAR, 2L)
        .with(ChronoField.DAY_OF_MONTH, 28L);
    System.out.println("Date after adjusting:  " + date3);    // 2024-02-28
  }
}
```

Output from the program:

Click here to view code image

```
Date-Time: 1945-08-06T08:15

Time of day:      08:15
Hour-of-day:      8
Minute-of-hour 1: 15
Minute-of-hour 2: 15
Second-of-minute: 0

Date:  1945-08-06
Year:  1945
Month value 1: 8
Month value 2: 8
Month-of-year: AUGUST
Day-of-week:   MONDAY
Day-of-month:  6

Ordering:
-1004-03-01 is before 1004-03-01: true
1004-03-01 is after -1004-03-01: true
-1004-03-01 is after -1004-03-01: false
-1004-03-01 is equal to 1004-03-01: false
-1004-03-01 is equal to -1004-03-01: true
1004 is a leap year: true

Using absolute adjusters:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28

Using temporal fields:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28
```

Temporal Arithmetic with Dates and Times

The temporal classes provide *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented by a specific amount of time*— for example, by number of hours or by number of months.

The `LocalTime` and `LocalDate` classes provide `plus/minus` methods to increment/ decrement a time or a date by a specific amount in terms of a *time unit* (e.g., hours, minutes, and seconds) or a *date unit* (e.g., years, months, and days), respectively. The `LocalDateTime` class provides `plus/minus` methods to increment/decrement a date-time object by an amount that is specified in terms of either a time unit or a date unit. For example, the `plusMonths(m)` and `plus(m, ChronoUnit.MONTHS)` method calls to a `LocalDate` object will return a new `LocalDate` object after adding the specified number of months passed as an argument to the method. Similarly, the `minusMinutes(mm)` and `minus(mm, ChronoUnit.MINUTES)` method calls on a `LocalTime` class will return a new `LocalTime` object after subtracting the specified number of minutes passed as an argument to the method. The change is relative, and is reflected in the new temporal object that is returned. Such `plus/minus` methods are also called *relative adjusters*, in contrast to *absolute adjusters* (p. 1035). The `ChronoUnit` enum type implements the `TemporalUnit` interface (p. 1044).

> Click here to view code image
>
> ```
> // LocalTime, LocalDateTime
> LocalTime/LocalDateTime minusHours(long hours)
> LocalTime/LocalDateTime plusHours(long hours)
>
> LocalTime/LocalDateTime minusMinutes(long minutes)
> LocalTime/LocalDateTime plusMinutes(long minutes)
>
> LocalTime/LocalDateTime minusSeconds(long seconds)
> LocalTime/LocalDateTime plusSeconds(long seconds)
>
> LocalTime/LocalDateTime minusNanos(long nanos)
> LocalTime/LocalDateTime plusNanos(long nanos)
> ```
>
> Return a copy of this `LocalTime` or `LocalDateTime` object with the specified amount either subtracted or added to the value of a specific time field. The calculation always wraps around midnight.
>
> For the methods of the `LocalDateTime` class, a `DateTimeException` is thrown if the result exceeds the date range.

Click here to view code image

```
// LocalDate, LocalDateTime
LocalDate/LocalDateTime minusYears(long years)
LocalDate/LocalDateTime plusYears(long years)

LocalDate/LocalDateTime minusMonths(long months)
LocalDate/LocalDateTime plusMonths(long months)

LocalDate/LocalDateTime minusWeeks(long weeks)
LocalDate/LocalDateTime plusWeeks(long weeks)

LocalDate/LocalDateTime minusDays(long days)
LocalDate/LocalDateTime plusDays(long days)
```

Return a copy of this `LocalDate` or `LocalDateTime` with the specified amount either subtracted or added to the value of a specific date field.

All methods throw a `DateTimeException` if the result exceeds the date range.

The first four methods will change the day of the month to the *last valid day* of the month if necessary, when the day of the month becomes invalid as a result of the operation.

The last four methods will adjust the month and year fields as necessary to ensure a valid result.

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(long amountToSub,
                                        TemporalUnit unit)
LocalTime/LocalDate/LocalDateTime plus(long amountToAdd, TemporalUnit unit)
boolean isSupported(TemporalUnit unit)
```

The `minus()` and `plus()` methods return a copy of this temporal object with the specified amount subtracted or added, respectively, according to the `TemporalUnit` specified. The `ChronoUnit` enum type implements the `TemporalUnit` interface, and its enum constants define specific temporal units (p. 1044).

The `isSupported()` method checks if the specified `TemporalUnit` is supported by this temporal object. It avoids an exception being thrown if it has been determined that the unit is supported.

The `minus()` or the `plus()` method can result in any one of these exceptions: `DateTimeException` (the amount cannot be subtracted or added), `Unsupported-TemporalTypeException` (unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(TemporalAmount amountToSub)
LocalTime/LocalDate/LocalDateTime plus(TemporalAmount amountToAdd)
```

Return a copy of this temporal object with the specified temporal amount subtracted or added, respectively. The classes `Period` (p. 1057) and `Duration` (p. 1064) implement the `TemporalAmount` interface.

The `minus()` or the `plus()` method can result in any one of these exceptions: `DateTimeException` (the temporal amount cannot be subtracted or added) or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalTime, LocalDate, LocalDateTime
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between two temporal objects in terms of the specified `TemporalUnit` (p. 1044). The start and the end points are this temporal object and the specified temporal argument `endExclusive`, where the end point is excluded. The result will be negative if the other temporal is before this temporal.

The `until()` method can result in any one of these exceptions: `DateTime-Exception` (the temporal amount cannot be calculated or the end temporal cannot be converted to the appropriate temporal object), `UnsupportedTemporalType-Exception` (unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Click here to view code image

```
// LocalDate
Period until(ChronoLocalDate endDateExclusive)
```

Calculates the amount of time between this date and another date as a `Period` (p. 1057). The calculation excludes the end date. The `LocalDate` class implements the `ChronoLocalDate` interface.

Example 17.3 demonstrates what we can call *temporal arithmetic*, where a `LocalDate` object is modified by adding or subtracting an amount specified as days, weeks, or months. Note how the value of the date fields is adjusted after each operation. In Example 17.3, the date `2021-10-23` is created at (1), and 10 months, 3 weeks, and 40 days are successively added to the new date object returned by each `plus` method call at (2), (3), and (4), respectively, resulting in the date `2022-10-23`. We then subtract 2 days, 4 weeks, and 11 months successively from the new date object returned by each `minus()` method call at (5), (6), and (7), respectively, resulting in the date `2021-10-23`. The method calls at (5), (6), and (7) are passed the temporal unit explicitly. In Example 17.3, several assignment statements are used to print the intermediate dates, but the code can be made more succinct by method chaining.

Click here to view code image

```
LocalDate date = LocalDate.of(2021, 10, 23);              // 2021-10-23
date = date.plusMonths(10).plusWeeks(3).plusDays(40);     // Method chaining
System.out.println(date);                                 // 2022-10-23
date = date.minus(2, ChronoUnit.DAYS)
           .minus(4, ChronoUnit.WEEKS)
           .minus(11, ChronoUnit.MONTHS);                 // Method chaining
System.out.println(date);                                 // 2021-10-23
```

The following code snippet illustrates the wrapping of time around midnight, as one would expect on a 24-hour clock. Each method call returns a new `LocalTime` object.

Click here to view code image

```
LocalTime witchingHour = LocalTime.MIDNIGHT               // 00:00
    .plusHours(14)                                        // 14:00
    .plusMinutes(45)                                      // 14:45
    .plusMinutes(30)                                      // 15:15
    .minusHours(15)                                       // 00:15
    .minusMinutes(15);                                    // 00:00
```

The next code snippet illustrates how the `plusYears()` method adjusts the day of the month, if necessary, when the year value is changed. The year in the date 2020-02-29 is changed to 2021 by adding 1 year, resulting in the following date: 2021-02-29. The `plusYears()` method adjusts the day of the month to the last valid day of the month, 28; as the year 2021 is not a leap year, the month of February cannot have 29 days.

Click here to view code image

```
LocalDate date5 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date5 = date5.plusYears(1);                   // Expected: 2021-02-29
System.out.println("Date5: " + date5);        // Adjusted: 2021-02-28
```

A temporal can also be adjusted by a *temporal amount*—for example, by a `Period` (p. 1057) or a `Duration` (p. 1064). The methods `plus()` and `minus()` accept the temporal amount as an argument, as shown by the code below.

Click here to view code image

```
LocalTime busDep = LocalTime.of(12, 15);                    // 12:15
Duration d1 = Duration.ofMinutes(30);                       // PT30M
LocalTime nextBusDep = busDep.plus(d1);                     // 12:45

LocalDate birthday = LocalDate.of(2020, 10, 23);            // 2020-10-23
Period p1 = Period.ofYears(1);                              // P1Y
LocalDate nextBirthday = birthday.plus(p1);                 // 2021-10-23
```

The `until()` method can be used to calculate the amount of time between two compatible temporal objects. The code below calculates the number of days to New Year's Day from the current date; the result, of course, will depend on the current date. In the call to the `until()` method at (1), the temporal unit specified is `ChronoUnit.DAYS`, as we want the difference between the dates to be calculated in days.

Click here to view code image

```
LocalDate currentDate = LocalDate.now();
LocalDate newYearDay = currentDate.plusYears(1).withMonth(1).withDayOfMonth(1)
long daysToNewYear = currentDate.until(newYearDay, ChronoUnit.DAYS); // (1)
System.out.println("Current Date: " + currentDate); // Current Date: 2021-03-0
System.out.println("New Year's Day: " + newYearDay);// New Year's Day: 2022-01
System.out.println("Days to New Year: " + daysToNewYear);// Days to New Year:
```

The statement at (1) below is meant to calculate the number of minutes until midnight from now, but throws a `DateTimeException` because it is not possible to obtain a `LocalDateTime` object from the end point, which is a `LocalTime` object.

Click here to view code image

```
long minsToMidnight = LocalDateTime.now()             // (1) DateTimeException
         .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

However, the statement at (2) executes normally, as both the start and end points are `LocalTime` objects.

Click here to view code image

```
long minsToMidnight = LocalTime.now()                 // (2)
         .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

Example 17.3 *Temporal Arithmetic*

Click here to view code image

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
```

```
public class TemporalArithmetic {

  public static void main(String[] args) {

    LocalDate date = LocalDate.of(2021, 10, 23);            // (1)
    System.out.println("Date:             " + date);        // 2021-10-23
    date = date.plusMonths(10);                             // (2)
    System.out.println("10 months after:  " + date);        // 2022-08-23
    date = date.plusWeeks(3);                               // (3)
    System.out.println("3 weeks after:    " + date);        // 2022-09-13
    date = date.plusDays(40);                               // (4)
    System.out.println("40 days after:    " + date);        // 2022-10-23

    date = date.minus(2, ChronoUnit.DAYS);                  // (5)
    System.out.println("2 days before:    " + date);        // 2022-10-21
    date = date.minus(4, ChronoUnit.WEEKS);                 // (6)
    System.out.println("4 weeks before:   " + date);        // 2022-09-23
    date = date.minus(11, ChronoUnit.MONTHS);               // (7)
    System.out.println("11 months before: " + date);        // 2021-10-23
  }
}
```

Output from the program:

```
Date:             2021-10-23
10 months after:  2022-08-23
3 weeks after:    2022-09-13
40 days after:    2022-10-23
2 days before:    2022-10-21
4 weeks before:   2022-09-23
11 months before: 2021-10-23
```

17.3 Using Temporal Units and Temporal Fields

Temporal units and temporal fields allow temporal objects to be accessed and manipulated in a human-readable way.

For temporal units and temporal fields supported by the `Period` and `Duration` classes, see §17.5, p. 1057, and §17.6, p. 1064, respectively.

Temporal Units

The `java.time.temporal.TemporalUnit` interface represents a *unit* of measurement, rather than an amount of such a unit—for example, the unit *years* to qualify that an amount of time should be interpreted as number of years. The `java.time.temporal.ChronoUnit` enum type implements this interface, defining the temporal units by constant names to provide convenient unit-based access to manipulate a temporal object. Constants defined by the `ChronoUnit` enum type include the following temporal units, among others: `SECONDS`, `MINUTES`, `HOURS`, `DAYS`, `MONTHS`, and `YEARS`.

The output from Example 17.4 shows a table with all the temporal units defined by the `ChronoUnit` enum type. It is not surprising that not all temporal units are valid for all types of temporal objects. The time units, such as `SECONDS`, `MINUTES`, and `HOURS`, are valid for temporal objects that are time-based, such as `LocalTime`, `LocalDateTime`, and `Instant`. Likewise, the date units, such as `DAYS`, `MONTHS`, and `YEARS`, are valid units for temporal objects that are date-based, such as `LocalDate`, `LocalDateTime`, and `ZonedDateTime`.

A `ChronoUnit` enum constant can be queried by the following selected methods:

```
Duration getDuration()
```

Gets the estimated duration of this unit in the ISO calendar system. For example, `ChronoUnit.DAYS.getDuration()` has the duration `PT24H` (i.e., 24 hours).

```
boolean isDateBased()
boolean isTimeBased()
```

Check whether this unit is a date unit or a time unit, respectively. For example, `ChronoUnit.HOURS.isDateBased()` is `false`, but `ChronoUnit.SECONDS.isTimeBased()` is `true`.

Click here to view code image

```
boolean isSupportedBy(Temporal temporal)
```

Checks whether this unit is supported by the specified temporal object. For example, `ChronoUnit.YEARS.isSupportedBy(LocalTime.MIDNIGHT)` is `false`.

Click here to view code image

```
static ChronoUnit[] values()
```

Returns an array containing the unit constants of this enum type, in the order they are declared. This method is called at (2) in Example 17.4.

The temporal classes provide the method `isSupported(unit)` to determine whether a temporal `unit` is valid for a temporal object. In Example 17.4, this method is used at (3), (4), and (5) to determine whether each temporal unit defined by the `ChronoUnit` enum type is a valid unit for the different temporal classes.

The following methods of the temporal classes all accept a temporal unit that qualifies how a numeric quantity should be interpreted:

- `minus(amount, unit)` and `plus(amount, unit)`, (p. 1040)

  Click here to view code image

  ```
  LocalDate date = LocalDate.of(2021, 10, 23);
  System.out.print("Date " + date);
  date = date.minus(10, ChronoUnit.MONTHS).minus(3, ChronoUnit.DAYS);
  System.out.println(" minus 10 months and 3 days: " + date);
  // Date 2021-10-23 minus 10 months and 3 days: 2020-12-20

  LocalTime time = LocalTime.of(14, 15);
  System.out.print("Time " + time);
  time = time.plus(9, ChronoUnit.HOURS).plus(70, ChronoUnit.MINUTES);
  System.out.println(" plus 9 hours and 70 minutes is " + time);
  // Time 14:15 plus 9 hours and 70 minutes is 00:25
  ```

- `until(temporalObj, unit)`, (p. 1040)

  Click here to view code image

  ```
  LocalDate fromDate = LocalDate.of(2021, 3, 1);
  LocalDate xmasDate = LocalDate.of(2021, 12, 25);
  long tilChristmas = fromDate.until(xmasDate, ChronoUnit.DAYS);
  System.out.println("From " + fromDate + ", days until Xmas: " + tilChri
  // From 2021-03-01, days until Xmas: 299
  ```

Temporal Fields

The `java.time.temporal.TemporalField` interface represents a specific field of a temporal object. The `java.time.temporal.ChronoField` enum type implements this interface, defining the fields by constant names so that a specific field can be conveniently accessed. Selected constants from the `ChronoField` enum type include `SECOND_OF_MINUTE`, `MINUTE_OF_DAY`, `DAY_OF_MONTH`, `MONTH_OF_YEAR`, and `YEAR`.

The output from Example 17.4 shows a table with all the temporal fields defined by the `ChronoField` enum type.

Analogous to a `ChronoUnit` enum constant, a `ChronoField` enum constant can be queried by the following selected methods:

```
TemporalUnit getBaseUnit()
```

Gets the unit that the field is measured in. For example, `ChronoField.DAY_OF_MONTH.getBaseUnit()` returns `ChronoUnit.DAYS`.

```
boolean isDateBased()
boolean isTimeBased()
```

Check whether this field represents a date or a time field, respectively. For example, `ChronoField.HOUR_OF_DAY.isDateBased()` is `false`, but `ChronoField.SECOND_ OF_MINUTE.isTimeBased()` is `true`.

Click here to view code image

```
boolean isSupportedBy(TemporalAccessor temporal)
```

Checks whether this field is supported by the specified temporal object. For example, `ChronoField.YEAR.isSupportedBy(LocalTime.MIDNIGHT)` is `false`.

```
static ChronoField[] values()
```

Returns an array containing the field constants of this enum type, in the order they are declared. This method is called at (7) in Example 17.4.

The temporal classes provide the method `isSupported(field)` to determine whether a temporal `field` is valid for a temporal object. In Example 17.4, this method is used at (8), (9), and (10) to determine whether each temporal field defined by the `ChronoField` enum type is a valid field for the different temporal classes.

The following methods of the temporal classes all accept a temporal field that designates a specific field of the temporal object:

- `get(field)`, (p. 1032)

  Click here to view code image

  ```
  LocalDate date = LocalDate.of(2021, 8, 13);
  int monthValue = date.get(ChronoField.MONTH_OF_YEAR);
  System.out.print("Date " + date + " has month of the year: " + monthValu
  // Date 2021-08-13 has month of the year: 8
  ```

- `with(field, amount)`, (p. 1035)

  Click here to view code image

  ```
  LocalDateTime dateTime = LocalDateTime.of(2021, 8, 13, 20, 20);
  System.out.print("Date-time " + dateTime);
  dateTime = dateTime.with(ChronoField.DAY_OF_MONTH, 11)
                     .with(ChronoField.MONTH_OF_YEAR, 1)
                     .with(ChronoField.YEAR, 2022);
  System.out.println(" changed to: " + dateTime);
  // Date-time 2021-08-13T20:20 changed to: 2022-01-11T20:20
  ```

In Example 17.4, the code at (1) and at (6) prints tables that show which `ChronoUnit` and `ChronoField` constants are valid in which temporal-based object. A `LocalTime` instance supports time-based units and fields, and a `LocalDate` instance supports date-based units and fields. A `LocalDateTime` or a `ZonedDateTime` supports both time-based and date-based units and fields. Using an invalid enum constant for a temporal object will invariably result in an `UnsupportedTemporalTypeException` being thrown.

Example 17.4 *Valid Temporal Units and Temporal Fields*

Click here to view code image

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class ValidTemporalUnitsAndFields {

  public static void main(String[] args) {

    // Temporals:
    LocalTime time = LocalTime.now();
    LocalDate date = LocalDate.now();
    LocalDateTime dateTime = LocalDateTime.now();
    ZonedDateTime zonedDateTime = ZonedDateTime.now();
    Instant instant = Instant.now();

    // Print supported units:                                           // (1
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoUnit", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoUnit[] units =  ChronoUnit.values();                          // (2
    for (ChronoUnit unit : units) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          unit.name(), time.isSupported(unit), date.isSupported(unit),  // (3
          dateTime.isSupported(unit), zonedDateTime.isSupported(unit),  // (4
          instant.isSupported(unit));                                   // (5
      }
    System.out.println();

    // Print supported fields:                                          // (6
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoField", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoField[] fields =  ChronoField.values();                       // (7
    for (ChronoField field : fields) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          field.name(), time.isSupported(field), date.isSupported(field),// (8
          dateTime.isSupported(field), zonedDateTime.isSupported(field), // (9
          instant.isSupported(field));                                  // (1
    }
    System.out.println();
  }
}
```

Output from the program (`ZDT` *stands for* `ZonedDateTime` *in the output*):

Click here to view code image

```
                   ChronoUnit LocalTime LocalDate LocalDateTime  ZDT  Instant
                       NANOS:    true     false       true      true    true
                      MICROS:    true     false       true      true    true
                      MILLIS:    true     false       true      true    true
                     SECONDS:    true     false       true      true    true
                     MINUTES:    true     false       true      true    true
                       HOURS:    true     false       true      true    true
                   HALF_DAYS:    true     false       true      true    true
                        DAYS:   false      true       true      true    true
                       WEEKS:   false      true       true      true   false
                      MONTHS:   false      true       true      true   false
                       YEARS:   false      true       true      true   false
                     DECADES:   false      true       true      true   false
                   CENTURIES:   false      true       true      true   false
                   MILLENNIA:   false      true       true      true   false
                        ERAS:   false      true       true      true   false
                     FOREVER:   false     false      false     false   false

                  ChronoField LocalTime LocalDate LocalDateTime  ZDT  Instant
              NANO_OF_SECOND:    true     false       true      true    true
                 NANO_OF_DAY:    true     false       true      true   false
             MICRO_OF_SECOND:    true     false       true      true    true
                MICRO_OF_DAY:    true     false       true      true   false
             MILLI_OF_SECOND:    true     false       true      true    true
                MILLI_OF_DAY:    true     false       true      true   false
            SECOND_OF_MINUTE:    true     false       true      true   false
               SECOND_OF_DAY:    true     false       true      true   false
              MINUTE_OF_HOUR:    true     false       true      true   false
               MINUTE_OF_DAY:    true     false       true      true   false
                HOUR_OF_AMPM:    true     false       true      true   false
          CLOCK_HOUR_OF_AMPM:    true     false       true      true   false
                 HOUR_OF_DAY:    true     false       true      true   false
           CLOCK_HOUR_OF_DAY:    true     false       true      true   false
                 AMPM_OF_DAY:    true     false       true      true   false
                 DAY_OF_WEEK:   false      true       true      true   false
ALIGNED_DAY_OF_WEEK_IN_MONTH:   false      true       true      true   false
 ALIGNED_DAY_OF_WEEK_IN_YEAR:   false      true       true      true   false
                DAY_OF_MONTH:   false      true       true      true   false

                 DAY_OF_YEAR:   false      true       true      true   false
                   EPOCH_DAY:   false      true       true      true   false
       ALIGNED_WEEK_OF_MONTH:   false      true       true      true   false
        ALIGNED_WEEK_OF_YEAR:   false      true       true      true   false
               MONTH_OF_YEAR:   false      true       true      true   false
             PROLEPTIC_MONTH:   false      true       true      true   false
                 YEAR_OF_ERA:   false      true       true      true   false
                        YEAR:   false      true       true      true   false
                         ERA:   false      true       true      true   false
             INSTANT_SECONDS:   false     false      false      true    true
              OFFSET_SECONDS:   false     false      false      true   false
```

17.4 Working with Instants

The temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` are suitable for representing human time in terms of year, month, day, hour, minute, second, and time zone. The `Instant` class can be used for representing computer time, specially *timestamps* that identify to a higher precision when an event occurred on the timeline. Instants are suitable for persistence purposes—for example, in a database.

An `Instant` represents *a point on the timeline*, measured with nanosecond precision from a starting point or origin which is defined to be at `1970-01-01T00:00:00Z`—that is, January 1, 1970, at midnight—and is called the *epoch*. Instants before the epoch have negative values, whereas instants after the epoch have positive values. The `Z` represents the time zone designator for the *zero UTC offset*, which is the time zone offset for all instants in the UTC standard (p. 1072). The text representation of the epoch shown above is in the ISO standard format used by the `toString()` method of the `Instant` class.

An `Instant` is modeled with two values:

- A `long` value to represent the *epoch-second*
- An `int` value to represent the *nano-of-second*

The nano-of-second must be a value in the range [0, 999999999]. This representation is reflected in the methods provided for dealing with instants. The `Instant` class shares many of the method name prefixes and the common method names in Table 17.2, p. 1026, and Table 17.3, p. 1026, with the other temporal classes, respectively. Although the `Instant` class has many methods analogous to the other temporal classes, as we shall see, there are also differences. `Instant` objects are, like objects of the other temporal classes, immutable and thread-safe.

Creating Instants

The `Instant` class provides the following predefined instants:

```
static Instant EPOCH
static Instant MAX
static Instant MIN
```

> These static fields of the `Instant` class define constants for the epoch (`1970-01-01T00:00:00Z`), the maximum (`1000000000-12-31T23:59:59.999999999Z`), and the minimum instants (`-1000000000-01-01T00:00Z`), respectively.

Following are selected methods for creating and converting instances of the `Instant` class:

```
static Instant now()
```

> Returns the current instant based on the system clock.
>
> Click here to view code image

```
static Instant ofEpochMilli(long epochMilli)
static Instant ofEpochSecond(long epochSecond)
static Instant ofEpochSecond(long epochSecond, long nanoAdjustment)
```

> These static factory methods return an `Instant` based on the millisecond, second, and nanosecond specified.
>
> Nanoseconds are implicitly set to zero. The argument values can be negative. Note that the amount is specified as a `long` value.

```
String toString()
```

> Returns a text representation of this `Instant`, such as `"2021-01-11T14:18:30Z"`. Formatting is based on the ISO instant format for date-time:
>
> Click here to view code image

```
uuuu-MM-ddTHH:mm:ss.SSSSSSSSSZ
```

> where `Z` designates the UTC standard (also known as *Coordinated Universal Time*).
>
> Click here to view code image

```
static Instant parse(CharSequence text)
```

> Returns an `Instant` parsed from a character sequence, such as `"2021-04-28T14:18:30Z"`, based on the ISO instant format. A `DateTimeParse-Exception` is thrown if the text cannot be parsed to an instant.
>
> Click here to view code image

```
ZonedDateTime atZone(ZoneId zone)
```

> Returns a `ZonedDateTime` by combining this instant with the specified time zone (p. 1072).

Analogous to the other temporal classes, the `Instant` class also provides the `now()` method to obtain the current instant from the system clock.

Click here to view code image

```
Instant currentInstant = Instant.now();      // 2021-03-09T10:48:01.914826Z
```

The `Instant` class provides the static factory method `ofEpoch`*`UNIT`*`()` to construct instants from seconds and nanoseconds. There is no method to construct an instant from just nanoseconds.

Click here to view code image

```
Instant inst1 = Instant.ofEpochMilli(-24L*60*60*1000);// Date 1 day before epo
Instant inst2 = Instant.ofEpochSecond(24L*60*60);     // Date 1 day after epoc
Instant inst3 = Instant.ofEpochSecond(24L*60*60 - 1,  // Date 1 day after epoc
                                      1_000_000_000L);
out.println("A day before: " + inst1); // Date 1 day before: 1969-12-31T00:00:
out.println("A day after:  " + inst2); // Date 1 day after : 1970-01-02T00:00:
out.println("A day after:  " + inst3); // Date 1 day after : 1970-01-02T00:00:
```

Note that the amount specified is a `long` value. The last statement above also illustrates that the nanosecond is adjusted so that it is always between 0 and 999,999,999. The adjustment results in the nanosecond being set to 0 and the second being incremented by 1.

The `toString()` method of the `Instant` class returns a text representation of an `Instant` based on the ISO standard. The code shows the text representation of the instant 500 nanoseconds after the epoch.

Click here to view code image

```
Instant inst4 = Instant.ofEpochSecond(0, 500);
out.println("Default format:  " + inst4);       // 1970-01-01T00:00:00.0000005
```

The `Instant` class also provides the `parse()` static method to create an instant from a string that contains a text representation of an instant, based on the ISO standard. Apart from treating the value of the nanosecond as optional, the method is strict in parsing the string. If the format of the string is not correct, a `DateTimeParseException` is thrown.

Click here to view code image

```
Instant instA = Instant.parse("1970-01-01T00:00:00.000000500Z");
Instant instB = Instant.parse("1949-03-01T12:30:15Z");
Instant instC = Instant.parse("-1949-03-01T12:30:15Z");
Instant instD = Instant.parse("-1949-03-01T12:30:15"); // DateTimeParseExcepti
```

The code below illustrates creating an `Instant` by combining a `LocalDateTime` object with a time zone offset. Three different zone-time offsets are specified at (2), (3), and (4) to convert the date-time created at (1) to an `Instant` on the UTC timeline, which has offset zero. Note that an offset ahead of UTC is subtracted and an offset behind UTC is added to adjust the values of the date/time from the `LocalDateTime` object to the UTC timeline.

Click here to view code image

```
LocalDateTime ldt = LocalDate.of(2021, 12, 25).atStartOfDay();  //(1)
Instant i1 = ldt.toInstant(ZoneOffset.of("+02:00"));     // (2) Ahead of UTC
Instant i2 = ldt.toInstant(ZoneOffset.UTC);              // (3) At UTC
Instant i3 = ldt.toInstant(ZoneOffset.of("-02:00"));     // (4) Behind UTC
System.out.println("ldt: " + ldt);
System.out.println("i1:  " + i1);
System.out.println("i2:  " + i2);
System.out.println("i3:  " + i3);
```

Output from the code:

```
ldt: 2021-12-25T00:00
i1:  2021-12-24T22:00:00Z
i2:  2021-12-25T00:00:00Z
i3:  2021-12-25T02:00:00Z
```

> Click here to view code image
>
> ```
> // LocalDateTime
> default Instant toInstant(ZoneOffset offset)
> ```
>
> Converts a date-time to an instant by combining this `LocalDateTime` object with the specified time zone. The valid offset in Java is in the range from –18 to +18 hours. The absolute value of the offset is added to or subtracted from the date-time depending on whether it is specified as a negative or positive value, respectively, keeping in mind that an `Instant` represents a point in time on the UTC timeline.
>
> This method is inherited by the `LocalDateTime` class from its superinterface `java.time.chrono.ChronoLocalDateTime`.

Accessing Temporal Fields in an Instant

The `Instant` class provides the following selected methods to access temporal fields in an instance of the class:

> ```
> int getNano()
> long getEpochSecond()
> ```
>
> Return the number of nanoseconds and the number of seconds represented by this instant from the start of the epoch, respectively. Note that the method names are without the `s` at the end.
>
> Click here to view code image
>
> ```
> int get(TemporalField field)
> long getLong(TemporalField field)
> ```
>
> The `get(field)` method will return the value of the specified `field` in this `Instant` as an `int`. Only the following `ChronoField` constants are supported: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS` (p. 1046). The first three fields will always return a valid value,

but the `INSTANT_SECONDS` field will throw a `DateTimeException` if the value does not fit into an `int`. All other fields result in an `UnsupportedTemporalTypeException`.

As the `getLong(field)` method returns the value of the specified `field` in this `Instant` as a `long`, there is no problem with overflow in returning a value designated by any of the four fields mentioned earlier.

Click here to view code image

```
boolean isSupported(TemporalField field)
```

The `isSupported(field)` determines whether the specified `field` is supported by this instant.

```
long toEpochMilli()
```

Returns the number of milliseconds that represent this `Instant` from the start of the epoch. The method throws an `ArithmeticException` in case of number overflow.

The code below shows how the `getNano()` and `getEpochSecond()` methods of the `Instant` class read the value of the nanosecond and the epoch-second fields of an `Instant` object, respectively.

Click here to view code image

```
Instant inst = Instant.ofEpochSecond(24L*60*60,    // 1 day and
                                     555_555_555L);// 555555555 ns after epoch
out.println(inst);                   // 1970-01-02T00:00:00.555555555Z
out.println(inst.getNano());         // 555555555 ns
out.println(inst.getEpochSecond());  // 86400 s
```

Reading the nanosecond and epoch-second fields of an `Instant` *in different units* can be done using the `get(field)` method. Note the value of the nanosecond field expressed in different units using `ChronoField` constants. To avoid a `DateTimeException` when number overflow occurs, the `getLong(field)` method is used instead of the `get(field)` method in accessing the epoch-second field.

Click here to view code image

```
out.println(inst.get(ChronoField.NANO_OF_SECOND));      // 555555555 ns
out.println(inst.get(ChronoField.MICRO_OF_SECOND));     // 555555 micros
out.println(inst.get(ChronoField.MILLI_OF_SECOND));     // 555 ms
out.println(inst.getLong(ChronoField.INSTANT_SECONDS)); // 86400 s
//out.println(inst.get(ChronoField.INSTANT_SECONDS));   // DateTimeException
//out.println(inst.get(ChronoField.HOUR_OF_DAY));       // UnsupportedTemporal
                                                        // TypeException
```

The `Instant` class provides the `toEpochMilli()` method to derive the position of the instant measured in milliseconds from the epoch; that is, the second and nanosecond fields are converted to milliseconds. Converting 1 day (86400 s) and 555555555 ns results in 86400555 ms.

Click here to view code image

```
out.println(inst.toEpochMilli());                        // 86400555 ms
```

Comparing Instants

The methods `isBefore()` and `isAfter()` can be used to determine if one instant is before or after the other on the timeline, respectively.

Click here to view code image

```
// instA is  1970-01-01T00:00:00.000000500Z
// instB is  1949-03-01T12:30:15Z
// instC is -1949-03-01T12:30:15Z
out.println(instA.isBefore(instB));                // false
out.println(instA.isAfter(instC));                 // true
```

The `Instant` class also overrides the `equals()` method and the `hashCode()` method of the `Object` class, and implements the `Comparable<Instant>` interface. Instants can readily be used in collections. The code below illustrates comparing instants.

Click here to view code image

```
out.println(instA.equals(instB));                   // false
out.println(instA.equals(instC));                   // false

List<Instant> list = Arrays.asList(instA, instB, instC);
Collections.sort(list);              // Natural order: position on the timeline
// [-1949-03-01T12:30:15Z, 1949-03-01T12:30:15Z, 1970-01-01T00:00:00.000000500
```

Click here to view code image

```
boolean isBefore(Instant other)
boolean isAfter(Instant other)
```

Determine whether this `Instant` is before or after the `other` instant on the timeline, respectively.

```
boolean equals(Object other)
```

Determines whether this `Instant` is equal to the `other` instant, based on the timeline position of the instants.

```
int hashCode()
```

> Returns a hash code for this `Instant`.
>
> `int compareTo(Instant other)`
>
> Compares this `Instant` with the `other` instant, based on the timeline position of the instants.

Creating Modified Copies of Instants

The `Instant` class provides the `with(field, newValue)` method that returns a copy of this instant with either the epoch-second or the nano-of-second set to a new value, while the other one is unchanged.

> Click here to view code image
>
> `Instant with(TemporalField field, long newValue)`
>
> Returns a copy of this instant where either the epoch-second or the nano-of-second is set to the specified value. The value of the other is retained.
>
> This method only supports the following `ChronoField` constants: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS` (p. 1046). For the first three fields, the nano-of-second is replaced by appropriately converting the specified value, and the epoch-second will be unchanged in the copy returned by the method. For the `INSTANT_SECONDS` field, the epoch-second will be replaced and the nanosecond will be unchanged in the copy returned by the method. Valid values that can be specified with these constants are [0–999999999], [0–999999], [0–999], and a `long`, respectively.
>
> This method throws a `DateTimeException` if the field cannot be set, an `Unsupported-TemporalTypeException` if the field is not supported, and an `ArithmeticException` if number overflow occurs.

In the code below, the three instants `i1`, `i2`, and `i3` will have the nano-of-second set to 5,000,000,000 nanoseconds using the `with()` method, but the epoch-second will not be changed.

Click here to view code image

```
Instant i0, i1, i2, i3;
i0 = Instant.now();
out.println(i0);                              // 2021-02-28T08:43:35.864Z
i1 = i0.with(ChronoField.NANO_OF_SECOND,  500_000_000);// 500000000 ns.
```

```
i2 = i0.with(ChronoField.MICRO_OF_SECOND, 500_000);    // 500000x1000 ns.
i3 = i0.with(ChronoField.MILLI_OF_SECOND, 500);        // 500x1000000 ns.
out.println(i1);                             // 2021-02-28T08:43:35.500Z

out.println(i1.equals(i2));                  // true
out.println(i1.equals(i3));                  // true
```

In the code below, `oneInstant` has the nano-of-second set to 500,000,000 nanoseconds and the epoch-second set to 1 day after the epoch.

Click here to view code image

```
Instant oneInstant = Instant.now()
                            .with(ChronoField.MILLI_OF_SECOND, 500)
                            .with(ChronoField.INSTANT_SECONDS, 24L*60*60);
out.println(oneInstant);                     // 1970-01-02T00:00:00.500Z
```

Temporal Arithmetic with Instants

The `Instant` class provides *plus* and *minus* methods that return a copy of the original instant that has been *incremented or decremented by a specific amount* specified in terms of either seconds, milliseconds, or nanoseconds. Each amounts below is explicitly designated as a `long` to avoid problems if the amount does not fit into an `int`.

Click here to view code image

```
Instant event =
    Instant.EPOCH                    //             1970-01-01T00:00:00Z
           .plusSeconds(7L*24*60*60) //  (+7days)   1970-01-08T00:00:00Z
           .plusSeconds(6L*60*60)    //  (+6hrs)    1970-01-08T06:00:00Z
           .plusSeconds(5L*60)       //  (+5mins)   1970-01-08T06:05:00Z
           .plusSeconds(4L)          //  (+4s)      1970-01-08T06:05:04Z
           .plusMillis(3L*100)       //  (+3ms)     1970-01-08T06:05:04.003Z
           .plusNanos(2L*1_000)      //  (+2micros) 1970-01-08T06:05:04.003002Z
           .plusNanos(1L);           //  (+1ns)     1970-01-08T06:05:04.0030020
```

However, it is more convenient to express the above calculation using the `plus(amount, unit)` method, which also allows the amount to be qualified by a unit. This is illustrated by the statement below, which is equivalent to the one above.

Click here to view code image

```
Instant ptInTime =
     Instant.EPOCH                           // 1970-01-01T00:00:00Z
            .plus(7L, ChronoUnit.DAYS)       // 1970-01-08T00:00:00Z
            .plus(6L, ChronoUnit.HOURS)      // 1970-01-08T06:00:00Z
            .plus(5L, ChronoUnit.MINUTES)    // 1970-01-08T06:05:00Z
            .plus(4L, ChronoUnit.SECONDS)    // 1970-01-08T06:05:04Z
            .plus(3L, ChronoUnit.MILLIS)     // 1970-01-08T06:05:04.003Z
            .plus(2L, ChronoUnit.MICROS)     // 1970-01-08T06:05:04.003002Z
```

```
                  .plus(1L, ChronoUnit.NANOS);    // 1970-01-08T06:05:04.003002001Z
```

The code below shows the `plus()` method of the `Instant` class that takes a `Duration` (p. 1064) as the amount to add.

Click here to view code image

```
Instant start = Instant.EPOCH
                       .plus(20, ChronoUnit.MINUTES);// 1970-01-01T00:20:00Z
Duration length = Duration.ZERO.plusMinutes(90);     // PT1H30M (90 mins)
Instant end = start.plus(length);                    // 1970-01-01T01:50:00Z
```

The `until()` method calculates the amount of time between two instants in terms of the unit specified in the method.

Click here to view code image

```
long eventDuration1 = start.until(end, ChronoUnit.MINUTES);   // 90 minutes
long eventDuration2 = start.until(end, ChronoUnit.HOURS);     // 1 hour
```

As an `Instant` does not represent an amount of time, but a point on the timeline, it cannot be used in temporal arithmetic with other temporal objects. Although an `Instant` incorporates a date, it is not possible to access it in terms of year and month.

Click here to view code image

```
Instant plusSeconds/minusSeconds(long seconds)
Instant plusMillis/minusMillis(long millis)
Instant plusNanos/minusNanos(long nanos)
```

Return a copy of this instant, with the specified amount added or subtracted. Note that the argument type is `long`.

The methods throw a `DateTimeException` if the result is not a valid instant, and an `ArithmeticException` if numeric flow occurs during the operation.

Click here to view code image

```
Instant plus(long amountToAdd, TemporalUnit unit)
Instant minus(long amountToSub, TemporalUnit unit)
```

Return a copy of this instant with the specified amount added or subtracted, respectively, where the specified `TemporalUnit` qualifies the amount (p. 1044).

The following units, defined as constants by the `ChronoUnit` class, can be used to qualify the amount: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044).

A method call can result in any one of these exceptions:

`DateTimeException` (if the operation cannot be performed), `UnsupportedTemporalTypeException` (if the unit is not supported), or `ArithmeticException` (if numeric overflow occurs).

Click here to view code image

```
Instant isSupported(TemporalUnit unit)
```

Returns `true` if the specified unit is supported (p. 1044), in which case, the unit can be used in plus/minus operations on an instant. If the specified unit is not supported, the plus/minus methods that accept a unit will throw an exception.

Click here to view code image

```
Instant plus(TemporalAmount amountToAdd)
Instant minus(TemporalAmount amountToSubtract)
```

Return a copy of this instant, with the specified amount added or subtracted. The amount is typically defined as a `Duration`.

A method call can result in any one of these exceptions: `DateTimeException` (if the operation cannot be performed) or `ArithmeticException` (if numeric overflow occurs).

Click here to view code image

```
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between two temporal objects in terms of the specified `TemporalUnit` (p. 1044). The start and end points are this temporal object and the specified temporal argument, where the end point is excluded.

The start point is an `Instant`, and the end point temporal is converted to an `Instant`, if necessary.

The following units, defined as constants by the `ChronoUnit` class, can be used to indicate the unit in which the result should be returned: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044).

The `until()` method can result in any one of these exceptions: `DateTimeException` (the temporal amount cannot be calculated or the end temporal cannot be converted to the appropriate temporal object), `UnsupportedTemporalTypeException` (the unit is not supported), or `ArithmeticException` (numeric overflow occurred).

Converting Instants

Each of the classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` provides the `ofInstant()` method to obtain a temporal object from an `Instant`. The code below shows how instants can be converted to other temporal objects for a given time zone. For date/time represented by this particular instant, the offset for the time zone `"America/New_York"` is -4 hours from UTC.

Click here to view code image

```
Instant instant = Instant.parse("2021-04-28T03:15:00Z");
ZoneId zid = ZoneId.of("America/New_York");
LocalTime lt = LocalTime.ofInstant(instant, zid);           // 10:18:30
LocalDate ld = LocalDate.ofInstant(instant, zid);           // 2021-04-27
LocalDateTime ldt = LocalDateTime.ofInstant(instant, zid);  // 2021-04-27T23:1
ZonedDateTime zdt = ZonedDateTime.ofInstant(instant, zid);
    // 2021-04-27T23:15-04:00[America/New_York]
```

> Click here to view code image
>
> ```
> static TemporalType ofInstant(Instant instant, ZoneId zone)
> ```
>
> Creates a *`TemporalType`* object from the given `Instant` and `ZoneId` (p. 1072), where *`TemporalType`* can be `LocalTime`, `LocalDate`, `LocalDateTime`, or `ZonedDateTime`.

17.5 Working with Periods

For representing *an amount of time*, the Date and Time API provides the two classes `Period` and `Duration`. We will concentrate on the `Period` class in this section and discuss the `Duration` class in §17.6, p. 1064.

The `Period` class essentially represents a *date-based amount of time* in terms of years, months, and days, whereas, the `Duration` class represents a *time-based amount of time* in terms of seconds and nanoseconds.

The date-based `Period` class can be used with the `LocalDate` class, and not surprisingly, the time-based `Duration` class can be used with the `LocalTime` class. Of course, the `LocalDateTime` class can use both temporal amount classes.

The `Period` and `Duration` classes are in the same package (`java.time`) as the temporal classes, and the repertoire of methods they provide should look familiar, as they share many of the method prefixes with the temporal classes (Table 17.2, p. 1026).

The mantra of immutable and thread-safe objects also applies to both the `Period` and the `Duration` classes.

Creating Periods

Like the temporal classes, the `Period` class does not provide any `public` constructors, but rather provides an overloaded static factory method `of()` to construct periods of different lengths, based on date units.

Click here to view code image

```
Period p = Period.of(2, 4, 8);          // (1)
System.out.println(p);                  // (2) P2Y4M8D (2 Years, 4 Months, 8 Da
Period p1 = Period.ofYears(10);         // P10Y, period of 10 years.
Period p2 = Period.ofMonths(14);        // P14M, period of 14 months.
Period p3 = Period.ofDays(40);          // P40D, period of 40 days.
Period p4 = Period.ofWeeks(2);          // P14D, period of 14 days (2 weeks).
```

The most versatile `of()` method requires the amount of time for all date units: years, months, and days, as at (1). Other `of()` methods create a period based on a particular date unit, as shown in the examples above.

The `toString()` method of the `Period` class returns a text representation of a `Period` according to the ISO standard: `P`*y*`Y`*m*`M`*d*`D`—that is, *y* Years, *m* Months, and *d* Days. The output from (2) above, `P2Y4M8D`, indicates a period of 2 years, 4 months, and 8 days.

The code snippet below does *not* create a period of 3 years, 4 months, and 5 days— it creates a period of only 5 days. The first method call is invoked with the class name, and the subsequent method calls are on the new `Period` object returned as a consequence of the previous call. The `of()` method creates a new `Period` object based on its argument.

Click here to view code image

```
Period period = Period.ofYears(3).ofMonths(4).ofDays(5); // P5D. Logical error
```

As we would expect, we can create a period that represents the amount of time between two dates by calling the static method `between()` of the `Period` class.

Click here to view code image

```
LocalDate d1 = LocalDate.of(2021, 3, 1);  // 2021-03-01
LocalDate d2 = LocalDate.of(2022, 3, 1);  // 2022-03-01
Period period12 = Period.between(d1, d2); // P1Y
Period period21 = Period.between(d2, d1); // P-1Y
```

The `Period` class also provides the static method `parse()` to create a period from a string that contains a text representation of a period in the ISO standard. If the format of the string is not correct, a `java.time.format.DateTimeParseException` is thrown.

Click here to view code image

```
Period period2 = Period.parse("P1Y15M20D"); // 1 year, 15 months, 20 days
Period period3 = Period.parse("P20D");      // 20 days
Period period4 = Period.parse("P5W");       // 35 days (5 weeks)
//  Period pX = Period.parse("P24H"); // java.time.format.DateTimeParseExcepti
```

```
static Period ZERO
```

This constant defines a `Period` of length zero (`P0D`).

Click here to view code image

```
static Period of(int years, int months, int days)
static Period ofYears(int years)
static Period ofMonths(int months)
static Period ofWeeks(int weeks)
static Period ofDays(int days)
```

These static factory methods return a `Period` representing an amount of time equal to the specified value of a date unit. Date units that are implicit are set to zero. A week is equal to 7 days. The argument value can be negative.

Click here to view code image

```
static Period between(LocalDate startDateInclusive,
                      LocalDate endDateExclusive)
```

This static method returns a `Period` consisting of the number of years, months, and days between the two dates. The calculation excludes the end date. The result of this method can be a negative period if the end date is before the start date.

```
String toString()
```

Returns a text representation of a `Period` according to the ISO standard. Typical formats are `P`*y*`Y`*m*`M`*d*`D` and `P`*n*`W`—that is, *y* Years, *m* Months, and *d* Days, or *n* Weeks.

Click here to view code image

```
static Period parse(CharSequence text)
```

This static method returns a `Period` parsed from a character sequence—for example, `"P3Y10M2D"` (3 years, 10 months, 2 days). A `java.time.format.Date-TimeParseException` is thrown if the text cannot be parsed to a period.

Accessing Date Units in a Period

The `Period` class provides the obvious `get`*`XXX`*`()` methods to read the values of date units of a `Period` object, where *`XXX`* can be `Years`, `Months`, or `Days`.

Click here to view code image

```
Period period5 = Period.of(2, 4, -10);
System.out.println("Period: " + period5);             // Period: P2Y4M-10D
System.out.println("Years:  " + period5.getYears());  // Years:  2
System.out.println("Months: " + period5.getMonths()); // Months: 4
System.out.println("Days:   " + period5.getDays());   // Days:   -10
```

Reading the value of date units of a `Project` object can also be achieved using the `get(unit)` method, where only the date units shown in the code below are allowed. A list of these valid temporal units can be obtained by calling the `getUnits()` method of the `Period` class.

The class also has methods to check if *any* date unit of a period has a negative value or if *all* date units of a period have the value zero.

Click here to view code image

```
System.out.println("Years:  " + period5.get(ChronoUnit.YEARS)); // Years:  2
System.out.println("Months: " + period5.get(ChronoUnit.MONTHS));// Months: 4
System.out.println("Days:   " + period5.get(ChronoUnit.DAYS));  // Days: -10
List<TemporalUnit> supportedUnits = period5.getUnits(); // [Years, Months, Day

System.out.println("Total months: " + period5.toTotalMonths()); // 28 months
System.out.println(period5.isNegative());                       // true
System.out.println(period5.isZero());                           // false
```

The class `Period` provides the method `toTotalMonths()` to derive the *total* number of months in a period. However, this calculation is solely based on the number of years and months in the period; the number of days is *not* considered. A `Period` just represents an amount of time, so it has no notion of a date. Conversion between months and years is not a problem, as 1 year is 12 months. However, conversion between the number of days and the other date units is problematic. The number of days in a year and in a month are very much dependent on whether the year is a leap year and on a particular month in the year, respectively. A `Period` is oblivious to both the year and the month in the year, as it represents an *amount* of time and *not* a *point* on the timeline.

```
int getYears()
int getMonths()
int getDays()
```

Return the value of a specific date unit of this period, indicated by the name of the method.

`long get(TemporalUnit unit)`

Returns the value of the specified `unit` in this `Period`. The only supported date `ChronoUnit`s are `YEARS`, `MONTHS`, and `DAYS` (p. 1044). All other units throw an exception.

`List<TemporalUnit> getUnits()`

Returns the list of date units supported by this period: `YEARS`, `MONTHS`, and DAYS (p. 1044). These date units can be used in the `get(TemporalUnit)` method.

`long toTotalMonths()`

Returns the total number of months in this period, based on the values of the years and months units. The value of the days unit is not considered.

`boolean isNegative()`

Determines whether the value of any date units of this period is negative.

`boolean isZero()`

Determines whether the values of all date units of this period are zero.

Comparing Periods for Equality

The `Period` class overrides the `equals()` and `hashCode()` methods of the `Object` class, but the class does *not* implement the `Comparable<E>` interface. The value of each date unit is compared individually, and must have the same value to be considered equal. A period of 1 year and 14 months is not equal to a period of 2 years and 2 months, or to a period of 26 months. For this reason, `Period` objects do not implement the `Comparable<E>` interface.

Click here to view code image

```
Period px = Period.of(1, 14, 0);
Period py = Period.of(2, 2, 0);
Period pz = Period.ofMonths(26);
System.out.println(px.equals(py));           // false
System.out.println(px.equals(pz));           // false
System.out.println(px.equals(Period.ZERO));  // false
```

`boolean equals(Object obj)`

> Determines whether this period is equal to another period, meaning that each corresponding date unit has the same value.
>
> `int hashCode()`
>
> Returns a hash code for this period.

Creating Modified Copies of Periods

The `Period` class provides `with` methods to set a new value for each date unit individually, while the values of the other date units remain unchanged. Note that each method call returns a new `Period` object, and chaining method calls work as expected.

Click here to view code image

```
Period p5 = Period.of(2, 1, 30) // P2Y1M30D
    .withYears(3)               // P3Y1M30D, sets the number of years
    .withMonths(16)             // P3Y16M30D, sets the number of months
    .withDays(1);               // P3Y16M1D, sets the number of days
```

> ```
> Period withYears(int years)
> Period withMonths(int months)
> Period withDays(int days)
> ```
>
> Return a copy of this period where a specific date unit is set to the value of the argument. The values of the other date units are not affected.

Temporal Arithmetic with Periods

The `Period` class provides *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented by a specific amount* specified in terms of a date unit—for example, as a number of years, months, or days. As the following code snippets show, only the value of a specific date unit is changed; the values of other date fields are unaffected. There is no implicit normalization performed, unless the `normalized()` method that normalizes only the months is called, adjusting the values of the months and years as necessary.

Click here to view code image

```
Period p6 = Period.of(2, 10, 30)  // P2Y10M30D
    .plusDays(10)                 // P2Y10M40D
    .plusMonths(8)                // P2Y18M40D
    .plusYears(1)                 // P3Y18M40D
    .normalized();                // P4Y6M40D
```

We can do simple arithmetic with periods. The code examples below use the `plus()` and the `minus()` methods of the `Period` class that take a `TemporalAmount` as an argument. Both the `Period` and the `Duration` classes implement the `TemporalAmount` interface. In the last assignment statement, we have shown the state of both new `Period` objects that are created.

Click here to view code image

```
Period p7 = Period.of(1, 1, 1);                  // P1Y1M1D
Period p8 = Period.of(2, 12, 30);                // P2Y12M30D
Period p9 = p8.minus(p7);                        // P1Y11M29D
p8 = p8.plus(p7).plus(p8);                       // P3Y13M31D, P5Y25M61D
```

Click here to view code image

```
Period plusYears/minusYears(long years)
Period plusMonths/minusMonths(long months)
Period plusDays/minusDays(long days)
```

Return a copy of this period, with the specified value for the date unit added or subtracted. The values of other date units are unaffected.

Click here to view code image

```
Period plus(TemporalAmount amount)
Period minus(TemporalAmount amount)
```

Return a copy of this period, with the specified temporal amount added or subtracted. The amount is of the interface type `TemporalAmount` that is implemented by the classes `Period` and `Duration`, but only `Period` is valid here. The operation is performed separately on each date unit. There is no normalization performed. A `DateTimeException` is thrown if the operation cannot be performed.

```
Period normalized()
```

Returns a copy of this period where the years and months are normalized. The number of days is not affected.

Click here to view code image

```
Period negated()
Period multipliedBy(int scalar)
```

Return a new instance of `Period` where the value of each date unit in this period is individually negated or multiplied by the specified `scalar`, respectively.

We can also do simple arithmetic with dates and periods. The following code uses the `plus()` and `minus()` methods of the `LocalDate` class that take a `TemporalAmount` as an argument (p. 1040). Note the adjustments performed to the month and the day fields to return a valid date in the last assignment statement.

Click here to view code image

```
Period p10 = Period.of(1, 1, 1);                    // P1Y1M1D
LocalDate date1 = LocalDate.of(2021, 3, 1);     // 2021-03-01
LocalDate date2 = date1.plus(p10);              // 2022-04-02
LocalDate date3 = date1.minus(p10);             // 2020-01-31
```

We can add and subtract periods from `LocalDate` and `LocalDateTime` objects, but not from `LocalTime` objects, as a `LocalTime` object has only time fields.

Click here to view code image

```
LocalTime time = LocalTime.NOON;
time = time.plus(p10);     // java.time.temporal.UnsupportedTemporalTypeExcept
```

Example 17.5 is a simple example to illustrate implementing period-based loops. The method `reserveDates()` at (1) is a stub for reserving certain dates, depending on the period passed as an argument. The `for(;;)` loop at (2) uses the `Local-Date.isBefore()` method to terminate the loop, and the `LocalDate.plus()` method to increment the current date with the specified period.

Example 17.5 *Period-Based Loop*

Click here to view code image

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodBasedLoop {
  public static void main(String[] args) {
    reserveDates(Period.ofDays(7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2021, 11, 20));
    System.out.println();
    reserveDates(Period.ofMonths(1),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 20));
    System.out.println();
    reserveDates(Period.of(0, 1, 7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 21));
  }

  public static void reserveDates(Period period,                  // (1)
                                  LocalDate fromDate,
                                  LocalDate toDateExclusive) {
    System.out.println("Start date: " + fromDate);
    for (LocalDate date = fromDate.plus(period);                  // (2)
         date.isBefore(toDateExclusive);
         date = date.plus(period)) {
```

```
      System.out.println("Reserved (" + period + "): " + date);
    }
    System.out.println("End date: " + toDateExclusive);
  }
}
```

Output from the program:

Click here to view code image

```
Start date: 2021-10-20
Reserved (P7D): 2021-10-27
Reserved (P7D): 2021-11-03
Reserved (P7D): 2021-11-10
Reserved (P7D): 2021-11-17
End date: 2021-11-20

Start date: 2021-10-20
Reserved (P1M): 2021-11-20
Reserved (P1M): 2021-12-20
End date: 2022-01-20

Start date: 2021-10-20
Reserved (P1M7D): 2021-11-27
Reserved (P1M7D): 2022-01-03
End date: 2022-01-21
```

We conclude this section with Example 17.6, which brings together some of the methods of the Date and Time API. Given a date of birth, the method `birthdayInfo()` at (1) calculates the age and the time until the next birthday. The age is calculated at (2) using the `Period.between()` method, which computes the period between two dates. The date for the next birthday is set at (3) as the birth date with the current year. The `if` statement at (4) adjusts the next birthday date by 1 year at (5), if the birthday has already passed. The statement at (6) calculates the time until the next birthday by calling the `LocalDate.until()` method. We could also have used the `Period.between()` method at (6). The choice between these methods really depends on which method makes the code more readable in a given context.

Example 17.6 *More Temporal Arithmetic*

Click here to view code image

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;

public class ActYourAge {

  public static void main(String[] args) {
    birthdayInfo(LocalDate.of(1981, Month.AUGUST, 19));
    birthdayInfo(LocalDate.of(1935, Month.JANUARY, 8));
  }
  public static void birthdayInfo(LocalDate dateOfBirth) {           // (1)
    LocalDate today = LocalDate.now();
    System.out.println("Today:         " + today);
    System.out.println("Date of Birth: " + dateOfBirth);
    Period p1 = Period.between(dateOfBirth, today);                  // (2)
    System.out.println("Age:           " +
                             p1.getYears()  + " years, " +
                             p1.getMonths() + " months, and " +
                             p1.getDays()    + " days");

    LocalDate nextBirthday =  dateOfBirth.withYear(today.getYear()); // (3)
    if (nextBirthday.isBefore(today) ||                              // (4)
        nextBirthday.isEqual(today)) {
      nextBirthday = nextBirthday.plusYears(1);                      // (5)
    }
    Period p2 = today.until(nextBirthday);                           // (6)
    System.out.println("Birthday in " + p2.getMonths() + " months and " +
                                    p2.getDays()    + " days");
  }
}
```

Possible output from the program:

Click here to view code image

```
Today:         2021-03-05
Date of Birth: 1981-08-19
Age:           39 years, 6 months, and 14 days
Birthday in 5 months and 14 days
Today:         2021-03-05
Date of Birth: 1935-01-08
Age:           86 years, 1 months, and 25 days
Birthday in 10 months and 3 days
```

17.6 Working with Durations

The `java.time.Duration` class implements a *time-based amount of time* in terms of *seconds* and *nanoseconds*, using a `long` and an `int` value for these time units, respectively. Although the `Duration` class models an amount of time in terms of seconds and nanoseconds, a duration can represent an amount of time in terms of days, hours, and minutes. As these time units have fixed lengths, it makes interoperability between these units possible. The time-based `Duration` class can be used with the `LocalTime` and `LocalDateTime` classes, as these classes have time fields. In contrast, the `Period` class essentially represents a *date-based amount of time* in terms of years, months, and days (p. 1057).

The `Period` and `Duration` classes are in the same package (`java.time`) as the temporal classes. The `Period` and the `Duration` classes provide similar methods, as they share many of the method prefixes and common methods with the temporal classes (Table 17.2, p. 1026, and Table 17.3, p. 1026). Their objects are immutable and thread-safe. However, there are also differences between the two classes (p. 1072). Familiarity with one would go a long way toward understanding the other.

Creating Durations

Like the `Period` class, the `Duration` class provides the static factory methods `of`*UNIT*`()` to construct durations with different units.

Click here to view code image

```
Duration d1 = Duration.ofDays(1L);                                   // PT24H
Duration d2 = Duration.ofHours(24L);                                 // PT24H
Duration d3 = Duration.ofMinutes(24L*60);                            // PT24H
Duration d4 = Duration.ofSeconds(24L*60*60);                         // PT24H
Duration d5 = Duration.ofMillis(24L*60*60*1000);                     // PT24H
Duration d6 = Duration.ofSeconds(24L*60*60 - 1, 1_000_000_000L); // (1) PT24H
Duration d7 = Duration.ofNanos(24L*60*60*1_000_000_000); // (2) PT24H
Duration d8 = Duration.ofNanos(24*60*60*1_000_000_000);  // (3) PT-1.857093632
```

The durations created above all have a length of 1 day, except for the one in the last declaration statement. Note that the amount specified should be a `long` value. It is a good defensive practice to always designate the amount as such in order to avoid inadvertent problems if the amount does not fit into an `int`. The designation `L` should be placed such that there is no danger of any previous operation in the expression causing a rollover. This problem is illustrated at (3), where the `int` value of the argument expression is rolled over, as it is greater than `Integer.MAX_VALUE`.

The statement at (1) above also illustrates that the value of the nanoseconds is adjusted so that it is always between 0 and 999,999,999. The adjustment at (1) results in the value 0 for nanoseconds and the number of seconds being incremented by 1.

Calling the `toString()` method on the first seven declarations above, the result is the string `"PT24H"` (a duration of 24 hours), whereas for the last duration at (3), the result string is `"PT-1.857093632S"`, which clearly indicates that the `int` amount was not interpreted as intended.

The previous declarations are equivalent to the ones below, where the amount is *qualified* with a specific unit in the call to the `of(value, unit)` method.

Click here to view code image

```
Duration d11 = Duration.of(1L, ChronoUnit.DAYS);                // P24H
Duration d22 = Duration.of(24L, ChronoUnit.HOURS);              // P24H
Duration d33 = Duration.of(24L*60, ChronoUnit.MINUTES);         // P24H
Duration d44 = Duration.of(24L*60*60, ChronoUnit.SECONDS);    // P24H
Duration d88 = Duration.of(24L*60*60*1000, ChronoUnit.MILLIS);// P24H
Duration d77 = Duration.of(24L*60*60*1_000_000_000,
                           ChronoUnit.NANOS);                 // P24H
```

The code snippet below does *not* create a duration of 8 days—it creates a duration of 24 hours. The first method call is invoked with the class name, and the subsequent method call is on the new `Duration` object returned as a consequence of the first call. The `of()` method creates a new `Duration` object based on its argument.

Click here to view code image

```
Duration duration = Duration.ofDays(7).ofHours(24);   // PT24H. Logical error.
```

Like the `Period` class, we can create a duration that represents the amount of time between two temporal objects by calling the static method `between()` of the `Duration` class.

Click here to view code image

```
LocalTime startTime = LocalTime.of(14, 30);                  // 14:30
LocalTime endTime   = LocalTime.of(17, 45, 15);              // 17:45:15
Duration interval1 = Duration.between(startTime, endTime);   // PT3H15M15S
Duration interval2 = Duration.between(endTime, startTime);   // PT-3H-15M-15S
```

Note the exception thrown in the last statement below because a `LocalDateTime` object *cannot* be derived from a `LocalTime` object, whereas the converse is true.

Click here to view code image

```
LocalDateTime dateTime = LocalDateTime.of(2021, 4, 28,
                                          17, 45, 15);    // 2021-04-28T17:45:
Duration interval3 = Duration.between(startTime, dateTime);  // PT3H15M15S
Duration interval4 = Duration.between(dateTime, startTime);  // DateTimeExcept
```

The `Duration` class also provides the `parse()` static method to create a duration from a text representation of a duration based on the ISO standard. If the format of the string is not correct, a `DateTimeParseException` is thrown. Formatting according to the `toString()` method is shown in parentheses.

Click here to view code image

```
Duration da = Duration.parse("PT3H15M10.1S");// 3hrs. 15mins. 10.1s.(PT3H15M10
Duration db = Duration.parse("PT0.999S");    // 999000000 nanos.    (PT0.999S)
Duration dc = Duration.parse("-PT30S");      // -30 seconds.        (PT-30S)
Duration dd = Duration.parse("P-24D");       // -24 days            (PT-576H)
Duration dd = Duration.parse("P24H");        // Missing T. DateTimeParseExcept
```

```
static Duration ZERO
```

This constant defines a `Duration` of length zero (`PT0S`).

Click here to view code image

```
static Duration ofDays(long days)
static Duration ofHours(long hours)
static Duration ofMinutes(long minutes)
static Duration ofMillis(long millis)
static Duration ofSeconds(long seconds)
static Duration ofSeconds(long seconds, long nanoAdjustment)
static Duration ofNanos(long nanos)
```

These static factory methods return a `Duration` representing an amount of time in seconds and nanoseconds that is equivalent to the specified amount, depending on the method. Nanoseconds are implicitly set to zero. The argument value can be negative. Standard definitions of the units are used. Note that the amount is specified as a `long` value.

Click here to view code image

```
static Duration of(long amount, TemporalUnit unit)
```

This static factory method returns a `Duration` representing an amount of time in seconds and nanoseconds that is equivalent to the specified amount in the specified temporal unit. The amount is specified as a `long` value, which can be negative.

Valid `ChronoUnit` constants to qualify the amount specified in the method call are the following: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044). These units have a standard or an estimated duration.

Click here to view code image

```
static Duration between(Temporal startInclusive, Temporal endExclusive)
```

This static method returns the duration between two temporal objects that must support the seconds unit and where it is possible to convert the second temporal argument to the first temporal argument type, if necessary. Otherwise, a `DateTimeException` is thrown. The result of this method can be a negative period if the end temporal is before the start temporal.

```
String toString()
```

Returns a text representation of a `Duration` according to the ISO standard: `PT`*h*`H`*m*`M`*d.d*`S`—that is, *h* Hours, *m* Minutes, and *d.d* Seconds, where the nanoseconds are formatted as a fraction of a second.

Click here to view code image

```
static Duration parse(CharSequence text)
```

> This static method returns a `Duration` parsed from a character sequence. The formats accepted are based on the ISO duration format `PTnHnMn.nS`—for example, `"PT2H3M4.5S"` (2 hours, 3 minutes, and 4.5 seconds). A `java.time.format.DateTimeParseException` is thrown if the text cannot be parsed to a duration.

Accessing Time Units in a Duration

The `Duration` class provides the `get`*UNIT*`()` methods to read the *individual* values of its time units. The class also has methods to check if the period has a negative value or if its value is zero.

Click here to view code image

```
Duration dx = Duration.ofSeconds(12L*60*60, 500_000_000L); // PT12H0.5S
out.println(dx.getNano());                                 // 500000000
out.println(dx.getSeconds());                              // 43200 (i.e. 12 h
```

Reading the *individual values* of time units of a `Duration` object can also be done using the `get(unit)` method, where only the `NANOS` and `SECONDS` units are allowed. A list of temporal units that are accepted by the `get(unit)` method can be obtained by calling the `getUnits()` of the `Duration` class.

Click here to view code image

```
out.println(dx.get(ChronoUnit.NANOS));       // 500000000
out.println(dx.get(ChronoUnit.SECONDS));     // 43200
out.println(dx.get(ChronoUnit.MINUTES));     // UnsupportedTemporalTypeExcepti
out.println(dx.getUnits());                  // [Seconds, Nanos]
```

The class `Duration` provides the method `to`*UNIT*`()` to derive the *total length* of the duration in the unit designated by the method name. The seconds and the nanoseconds are converted to this unit, if necessary.

Click here to view code image

```
out.println("Days:    " + dx.toDays());      // Days:    0
out.println("Hours:   " + dx.toHours());     // Hours:   12
out.println("Minutes: " + dx.toMinutes());   // Minutes: 720
out.println("Millis:  " + dx.toMillis());    // Millis:  43200500
out.println("Nanos:   " + dx.toNanos());     // Nanos:   43200500000000
```

```
int getNano()
long getSeconds()
```

Return the number of nanoseconds and seconds in this duration, respectively—*not* the total length of the duration. Note that the first method name is `getNano`, without the `s`.

```
long get(TemporalUnit unit)
```

Returns the value of the specified `unit` in this `Duration`—*not* the total length of the duration. The only supported `ChronoUnit` constants are `NANOS` and `SECONDS` (p. 1044). Other units result in an `UnsupportedTemporalTypeException`.

```
List<TemporalUnit> getUnits()
```

Returns the list of time units supported by this duration: `NANOS` and `SECONDS` (p. 1044). These time units can be used with the `get(unit)` method.

```
long toDays()
long toHours()
long toMinutes()
long toMillis()
long toNanos()
```

Return the *total* length of this duration, converted to the unit designated by the method, if necessary. Note that there is no `toSeconds()` method. Also, the method name is `toNanos`—note the `s` at the end.

The methods `toMillis()` and `toNanos()` throw an `ArithmeticException` in case of number overflow.

```
boolean isNegative()
```

Determines whether the total length of this duration is negative.

```
boolean isZero()
```

Determines whether the total length of this duration is zero.

Comparing Durations

The `Duration` class overrides the `equals()` method and the `hashCode()` method of the `Object` class, and implements the `Comparable<Duration>` interface. Durations can readily be used in collections. The code below illustrates comparing durations.

Click here to view code image

```
Duration eatBreakFast = Duration.ofMinutes(20L);                    // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);                 // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES);       // PT45M
```

```
out.println(eatBreakFast.equals(eatLunch));                   // false
out.println(Duration.ofSeconds(0).equals(Duration.ZERO));      // true

List<Duration> ld = Arrays.asList(eatSupper, eatBreakFast, eatLunch );
Collections.sort(ld);                                  // Natural order.
out.println(ld);                                       // [PT20M, PT30M, PT45M]
```

Click here to view code image

```
boolean equals(Object otherDuration)
```

Determines whether the *total length* of this duration is equal to the total length of the other duration.

```
int hashCode()
```

Returns a hash code for this duration.

Click here to view code image

```
int compareTo(Duration otherDuration)
```

Compares the *total length* of this duration to the total length of the other duration.

Creating Modified Copies of Durations

The `Duration` class provides `with`*`UNIT`*`()` methods to set a new value for each time unit individually, while the value of the other time unit is retained. Note that each method call returns a new `Duration` object, and chaining method calls works as expected.

Click here to view code image

```
Duration oneDuration =  Duration.ZERO                      // PT0S
                                .withNanos(500_000_000)    // New copy: PT0.5S
                                .withSeconds(12L*60*60);   // New copy: PT12H0.
```

Click here to view code image

```
Duration withNanos(int nanoOfSecond)
Duration withSeconds(long seconds)
```

Return a copy of this duration where either the nanosecond or the seconds are set to the value of the argument, respectively. The value of the other time unit is retained.

Temporal Arithmetic with Durations

The `Duration` class provides *plus* and *minus* methods that return a copy of the original object that has been *incremented or decremented* *by a specific amount* specified in terms of a unit—for example, as a number of days, hours, minutes, or seconds.

Click here to view code image

```
Duration max20H = Duration.ZERO                           // PT0S
                          .plusHours(10)                  // PT10H
                          .plusMinutes(10*60 + 30)        // PT20H30M
                          .plusSeconds(6*60*60 + 15)      // PT26H30M15S
                          .minusMinutes(2*60 + 30)        // PT24H15S
                          .minusSeconds(15);              // PT24H
```

The `plus()` and the `minus()` methods also allow the amount to be qualified by a unit that has a standard or an estimated duration, as illustrated by the statement below, which is equivalent to the one above.

Click here to view code image

```
Duration max20H2 =
    Duration.ZERO                                         // PT0S
            .plus(10L,           ChronoUnit.HOURS)        // PT10H
            .plus(10*60 + 30,    ChronoUnit.MINUTES)      // PT20H30M
            .plus(6*60*60L + 15, ChronoUnit.SECONDS)      // PT26H3015S
            .minus(2*60 + 30,    ChronoUnit.MINUTES)      // PT24H15S
            .minus(15,           ChronoUnit.SECONDS);     // PT24H
```

The code below shows the `plus()` and the `minus()` methods of the `Duration` class that take a `Duration` as the amount to add or subtract.

Click here to view code image

```
Duration eatBreakFast = Duration.ofMinutes(20L);                // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);             // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES);   // PT45M
Duration totalTimeForMeals = eatBreakFast                       // PT20M
    .plus(eatLunch)                                             // PT50M
    .plus(eatSupper);                                           // PT1H35M
```

The statement below shows other arithmetic operations on durations and how they are carried out, together with what would be printed if the intermediate results were also written out.

Click here to view code image

```
Duration result = Duration.ofSeconds(-100, -500_000_000) // -100.5 => PT-1M-40
                          .abs()           // abs(-100.5) = 100.5  => PT1M40.5
                          .multipliedBy(4) // 100.5*4 = 402 => PT6M42S
                          .dividedBy(2);   // 402 / 2 = 201 => PT3M21S
```

Click here to view code image

```
Duration plusDays/minusDays(long days)
Duration plusHours/minusHours(long hours)
Duration plusMinutes/minusMinutes(long minutes)
Duration plusSeconds/minusSeconds(long seconds)
Duration plusMillis/minusMillis(long millis)
Duration plusNanos/minusNanos(long nanos)
```

Return a copy of this duration, with the specified value of the unit designated by the method name added or subtracted, but converted first to seconds, if necessary. Note that the argument type is `long`.

Click here to view code image

```
Duration plus(long amountToAdd, TemporalUnit unit)
Duration minus(long amountToSub, TemporalUnit unit)
```

Return a copy of this duration with the specified amount added or subtracted, respectively, according to the `TemporalUnit` specified (p. 1044).

Valid `ChronoUnit` constants to qualify the amount specified in the method call are the following: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS` (p. 1044). These units have a standard or an estimated duration. Other units result in an `UnsupportedTemporalTypeException`.

Click here to view code image

```
Duration plus(Duration duration)
Duration minus(Duration duration)
```

Return a copy of this duration, with the specified duration added or subtracted.

```
Duration abs()
```

Returns a copy of this duration with a positive length.

```
Duration negated()
```

Returns a copy of this duration where the length has been negated.

Click here to view code image

```
Duration dividedBy(long divisor)
Duration multipliedBy(long multiplicand)
```

The first method returns a new instance with the result of dividing the length of this duration by the specified `divisor`. Division by zero would bring down untold calamities.

The second method returns a new instance with the result of multiplying the length of this duration by the specified `multiplicand`.

We can perform arithmetic operations on durations and temporal objects. The following code uses the `plus()` and `minus()` methods of the `LocalTime` and `LocalDateTime` classes that take a `TemporalAmount` as an argument (p. 1040). We can add and subtract durations from `LocalTime` and `LocalDateTime` objects, but not from `LocalDate` objects, as a `LocalDate` object only supports date units.

Click here to view code image

```
LocalTime timeA = LocalTime.of(14,45,30);                 // 14:45:30
LocalDate dateA = LocalDate.of(2021, 4, 28);              // 2021-04-28
LocalDateTime dateTimeA = LocalDateTime.of(dateA, timeA); // 2021-04-28T14:45:

Duration amount = Duration.ofMinutes(20);                 // PT20M

timeA = timeA.plus(amount);                               // 15:05:30
dateTimeA = dateTimeA.minus(amount);                      // 2021-04-28T14:25:

dateA = dateA.minus(amount);                 // UnsupportedTemporalTypeExcepti
```

Example 17.7 illustrates implementing duration-based loops. The program prints the showtimes, given when the first show starts, the duration of the show, and when the theatre closes. The `for(;;)` loop at (1) uses the `LocalTime.isBefore()` method and the `LocalTime.plus(duration)` method to calculate the showtimes.

Example 17.7 *Duration-Based Loop*

Click here to view code image

```
import java.time.LocalTime;
import java.time.Duration;

public class DurationBasedLoop {
  public static void main(String[] args) {
    Duration duration = Duration.ofHours(2).plusMinutes(15);     // PT2H15M
    LocalTime firstShowTime = LocalTime.of(10, 10);              // 10:10
    LocalTime endTimeExclusive = LocalTime.of(23, 0);            // 23:00
    for (LocalTime time = firstShowTime;                         // (1)
         time.plus(duration).isBefore(endTimeExclusive);
         time = time.plus(duration)) {
      System.out.println("Showtime (" + duration + "): " + time);
    }
    System.out.println("Closing time: " + endTimeExclusive);
  }
}
```

Output from the program:

```
Showtime (PT2H15M): 10:10
Showtime (PT2H15M): 12:25
Showtime (PT2H15M): 14:40
Showtime (PT2H15M): 16:55
Showtime (PT2H15M): 19:10
Closing time: 23:00
```

Differences between Periods and Durations

Table 17.4 summarizes the differences between selected methods of the `Period` and the `Duration` classes, mainly in regard to the temporal units supported, representation for parsing and formatting, and comparison. *N/A* stands for *Not Applicable*.

Table 17.4 *Some Differences between the* `Period` *Class and the* `Duration` *Class*

Methods	The `Period` class	The `Duration` class
`of(amount, unit)`	N/A	Valid `ChronoUnits`: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, `DAYS` (p. 1065).
`parse(text)` `toString()`	Representation based on: `PnYnMnD` and `PnW` (p. 1057).	`PnDTnHnMn.nS` (p. 1065).
`get(unit)`	Supported `ChronoUnits`: `YEARS`, `MONTHS`, `DAYS` (p. 1059).	Supported `ChronoUnits`: `NANOS`, `SECONDS` (p. 1067).
`getUnits()`	Supported `ChronoUnits`: `YEARS`, `MONTHS`, `DAYS` (p. 1059).	Supported `ChronoUnits`: `NANOS`, `SECONDS` (p. 1067).
`equals(other)`	Based on values of individual units (p. 1059).	Based on total length (p. 1067).
`compareTo(other)`	N/A	Natural order: total length (p. 1067).
`minus(amount, unit)` `plus(amount, unit)`	N/A	Valid `ChronoUnits`: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, `DAYS` (p. 1069).
`abs()`	N/A	Returns copy with positive length (p. 1069).
`dividedBy(divisor)`	N/A	Returns copy after dividing by divisor (p. 1069).
`normalized()`	Only years and months normalized (p. 1061).	N/A

17.7 Working with Time Zones and Daylight Savings

The following three classes in the `java.time` package are important when dealing with date and time in different time zones and daylight saving hours: `ZoneId`, `ZoneOffset`, and `ZonedDateTime`.

UTC (*Coordinated Universal Time*) is the primary *time standard* used for keeping time around the world. *UTC/Greenwich* is the time at Royal Observatory, Greenwich, England. It is the basis for defining time in different regions of the world.

Time Zones and Zone Offsets

A *time zone* defines a region that observes the same standard time. The time observed in a region is usually referred to as *the local time*. A time zone is described by a *zone offset from UTC/Greenwich* and any *rules* for applying daylight saving time (DST). Time zones that practice DST obviously have variable offsets during the year to account for DST.

In Java, each time zone has a zone ID that is represented by the class `java.time.ZoneId`. The class `java.time.ZoneOffset`, which extends the `ZoneId` class, represents a zone offset from UTC/Greenwich. For example, the time zone with `US/Eastern` as the zone ID has the offset `-04:00` during daylight saving hours—that is, it is 4 hours behind UTC/Greenwich when DST is in effect.

The time zone offset at UTC/Greenwich is represented by `ZoneOffset.UTC` and, by convention, is designated by the letter `Z`. GMT (*Greenwich Mean Time*) has zero offset from UTC/Greenwich (`UTC+0`), thus the two are often used as synonyms; for example, `GMT-4` is equivalent to `UTC-4`. However, GMT is a time zone, whereas UTC is a time standard.

Java uses the IANA Time Zone Database (TZDB) maintained by the Internet Assigned Numbers Authority (IANA) that updates the database regularly, in particular, regarding changes to the rules for DST practiced by a time zone (p. 1073).

A set with names of available time zones can readily be obtained by calling the `ZoneId.getAvailableZoneIds()` method. Time zones have unique names of the form *`Area/Location`*—for example, `US/Eastern`, `Europe/Oslo`. The following code prints a very long list of time zones that are available to a Java application.

Click here to view code image

```
ZoneId.getAvailableZoneIds()                    // Prints a long list of zone nam
      .stream()
      .sorted()
      .forEachOrdered(System.out::println);   // Output not shown intentionally
```

The `ZoneId.of()` method creates an appropriate zone ID depending on the format of its argument:

- *UTC-equivalent ID*, if only `"Z"`, `"UTC"`, or `"GMT"` is specified. As these designations are equivalent, the result is `ZoneOffset.UTC`; that is, it represents the offset `UTC+0`.
- *Offset-based ID*, if the format is `"+hh:mm"` or `"-hh:mm"`—for example, `"+04:00"`, `"-11:30"`. The result is an instance of the `ZoneOffset` class with the parsed offset.
- *Prefix offset-based ID*, if the format is the prefix `UTC` or `GMT` followed by a numerical offset—for example, `"UTC+04:00"`, `"GMT-04:00"`. The result is a time zone represented by a `ZoneId` with the specified prefix and a parsed offset.

- *Region-based ID*, if the format is "*Area/Location*"—for example, `"US/Eastern"`, `"Europe/Oslo"`. The result is a `ZoneId` that can be used, among other things, to look up the underlying zone rules associated with the zone ID. In the examples in this section, a zone ID is specified in the format of a region-based ID.

The code below creates a region-based zone ID. The method `ZoneId.systemDefault()` returns the system default zone ID.

Click here to view code image

```
ZoneId estZID = ZoneId.of("US/Eastern");                // Create a time zone ID
System.out.println(estZID);                             // US/Eastern
System.out.println(ZoneId.systemDefault());             // Europe/Oslo
```

Selected methods in the `ZoneId` abstract class are presented below. The concrete class `ZoneOffset` extends the `ZoneId` class.

Click here to view code image

```
static ZoneId of(String zoneId)
```

Returns an appropriate zone ID depending on the format of the zone ID string. See the previous discussion on zone ID.

Click here to view code image

```
String          toString()
abstract String getId()
```

Return a string with the zone ID, typically in one of the formats accepted by the `of()` method.

```
abstract ZoneRules getRules()
```

Retrieves the associated time zone rules for this zone ID. The rules determine the functionality associated with a time zone, such as daylight savings (p. 1082).

Click here to view code image

```
static Set<String> getAvailableZoneIds()
```

Returns a set with the available time zone IDs.

```
static ZoneId systemDefault()
```

Returns the zone ID of the default time zone of the system.

The `ZonedDateTime` Class

The `ZonedDateTime` class represents a date-time with time zone information, and an instance of the class is referred to as a *zoned date-time* (Table 17.1, p. 1025). The date and time fields are stored with nanosecond precision. The time zone information is represented by a time zone with a zone offset from UTC/Greenwich. In essence, a zoned date-time represents an instant on a specific timeline determined by its zone ID and its zone offset from UTC/Greenwich. This timeline is referred to as the *local timeline*, whereas the timeline at UTC/Greenwich is referred to as the *instant timeline*.

A majority of the methods in the `ZonedDateTime` class should be familiar from the temporal classes discussed earlier in this chapter (p. 1027). Both Table 17.2, p. 1026, and Table 17.3, p. 1026, showing the common method name prefixes and common methods, apply to the `ZonedDateTime` class as well. The `LocalDateTime` class is the closest equivalent of the `ZonedDateTime` class.

Issues relating to daylight savings are covered later in this section (p. 1082). In particular, the methods `of()`, `with()`, `minus()`, and `plus()` take into consideration daylight savings.

The methods `get()`, `with()`, `plus()`, `minus()`, and `until()` throw an `UnsupportedTemporalTypeException` when an unsupported field is accessed, and a `DateTimeException` when the operation cannot be performed for any other reason.

Objects of the `ZonedDateTime` class are also immutable and thread-safe. The class also overrides the `equals()` and the `hashCode()` methods of the `Object` class, but in contrast to the `LocalDateTime` class, its objects are *not* comparable.

With that overview, we turn our attention to dealing with zoned date-times as represented by the `ZonedDateTime` class.

Creating Zoned Date-Time

Analogous to the other temporal classes, the current zoned date-time can be obtained from the system clock in either the default time zone or a specific time zone by calling the `now()` method of the `ZonedDateTime` class. The zoned date-times created at (1a), (2a), and (3a) represent the same date-time locally in the default time zone `Europe/Oslo`, at UTC/Greenwich (`UTC`), and in the `US/Eastern` time zone, respectively.

The text representation of the zoned date-times from (1a), (2a), and (3a) is shown below at (1b), (2b), and (3b), respectively. The text representation of a zoned date-time returned by the `toString()`

method shows the date and the time separated by the letter `T`, followed by the zone offset and the time zone.

Click here to view code image

```
ZonedDateTime defaultZDT = ZonedDateTime.now();                          // (1
ZonedDateTime utcZDT     = ZonedDateTime.now(ZoneId.of("UTC"));          // (2
ZonedDateTime edtZDT     = ZonedDateTime.now(ZoneId.of("US/Eastern"));   // (3
System.out.println("Default Zone Date-time:  " + defaultZDT);
System.out.println("UTC Date-time:           " + utcZDT);
System.out.println("EDT Zone Date-time:      " + edtZDT);
```

Output from the print statements:

Click here to view code image

```
Default Zone Date-time: 2021-07-11T11:35:20.008+02:00[Europe/Oslo]       // (1
UTC Date-time:          2021-07-11T09:35:20.023Z[UTC]                    // (2
EDT Zone Date-time:     2021-07-11T05:35:20.023-04:00[US/Eastern]        // (3
```

The local time in the `Europe/Oslo` time zone has the zone offset `+02:00`—meaning it is 2 hours *ahead* of UTC/Greenwich, as it is east of Greenwich. The date-time at UTC/Greenwich (`UTC`) has no zone offset, just the letter `Z` instead of an offset. The local time in the `US/Eastern` time zone has the zone offset `-04:00`—meaning it is 4 hours *behind* UTC/Greenwich, as it is west of Greenwich. On this particular date, the time difference between the `Europe/Oslo` time zone and the `US/Eastern` time zone is 6 hours, where daylight saving time is in effect in both time zones (p. 1082).

One convenient way to create a zoned date-time is to assemble it from its constituent parts. We will use the following declarations to create zoned date-times.

Click here to view code image

```
LocalTime concertTime = LocalTime.of(00, 10);                  // 00:10
LocalDate concertDate = LocalDate.of(1973, Month.JANUARY, 14); // 1973-01-14
LocalDateTime concertDT = LocalDateTime.of(concertDate,
                                           concertTime);   // 1973-01-14T00:10
ZoneId hwZID = ZoneId.of("US/Hawaii");                     // US/Hawaii
Instant instantZ = Instant.ofEpochSecond(95854200);        // 1973-01-14T10:10:
```

The code below creates three zoned date-times from constituent parts. The arguments in the method call comprise the parts of a zoned date-time in each case. We note that the zone offset of the `US/Hawaii` time zone is `-10:00`—that is, 10 hours behind UTC/Greenwich. Note how the `ofInstant()` method converts the time in the instant to the correct local time in the specified time zone.

Click here to view code image

```
ZonedDateTime concertZDT0 = ZonedDateTime.of(concertDate, concertTime, hwZID);
ZonedDateTime concertZDT1 = ZonedDateTime.of(concertDT, hwZID);
ZonedDateTime concertZDT2 = ZonedDateTime.ofInstant(instantZ, hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
boolean areEqual = concertZDT0.equals(concertZDT1)
                && concertZDT0.equals(concertZDT2);      // true
```

The `LocalDateTime` and the `Instant` classes also provide the `atZone()` method that combines a date-time or an instant, respectively, with a time zone to create a zoned date-time. The two zoned date-times created below are equal to the three created earlier.

Click here to view code image

```
ZonedDateTime concertZDT3 = concertDT.atZone(hwZID);
ZonedDateTime concertZDT4 = instantZ.atZone(hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
```

We can also use the `parse()` method to create a zoned date-time from a text string that is compatible with the ISO format (p. 1129).

Click here to view code image

```
ZonedDateTime concertZDT5
    = ZonedDateTime.parse("1973-01-14T00:10-10:00[US/Hawaii]");
```

Click here to view code image

```
static ZonedDateTime now()
static ZonedDateTime now(ZoneId zone)
```

Return a `ZonedDateTime` containing the current date-time from the system clock in the default time zone or in the specified time zone, respectively.

Click here to view code image

```
static ZonedDateTime of(int year, int month, int dayOfMonth, int hour,
                        int minute, int second, int nanoOfSecond, ZoneId zone)
static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)
static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
static ZonedDateTime ofInstant(Instant instant, ZoneId zone)
```

Return a `ZonedDateTime` formed from the specified arguments. Note the `zone` argument that is required. The methods throw a `DateTimeException` when the operation cannot be performed for some reason.

Note that these methods take into consideration any adjustments because of daylight savings (p. 1082).

Click here to view code image

```
static ZonedDateTime parse(CharSequence text)
```

Returns a `ZonedDateTime` by parsing the text string according to `DateTimeFormatter.ISO_ZONED_DATE_TIME` (p. 1129)—for example, `"2021-07-03T16:15:30+01:00 [Europe/Oslo]"`.

```
String toString()
```

Returns a string comprising the text representation of the constituent parts: the `LocalDateTime`, typically followed by the zone offset and the time zone—for example, `2021-07-11T05:35:20.023-04:00[US/Eastern]`.

Accessing Fields of Zoned Date-Time

The `ZonedDateTime` class provides many `get` methods to access the values of various fields of a zoned date-time. The most versatile is the `get(field)` method, which supports all the constants defined by the `ChronoField` enum type (p. 1046). In addition, there are get methods for specific fields, and methods for extracting the different constituent parts of a zoned date-time.

Click here to view code image

```
// Using ChronoField constants:
int theDay = concertZDT0.get(ChronoField.DAY_OF_MONTH);         // 14
int theMonthValue = concertZDT0.get(ChronoField.MONTH_OF_YEAR);  // 1
int theYear = concertZDT0.get(ChronoField.YEAR);                 // 1973

// Using specific get methods:
int theMonthValue2 = concertZDT0.getMonthValue();                // 1
Month theMonth     = concertZDT0.getMonth();                     // JANUARY

// Extracting constituent parts:
LocalTime theTime     = concertZDT0.toLocalTime();         // 00:10
LocalDate theDate     = concertZDT0.toLocalDate();         // 1973-01-14
LocalDateTime theDT   = concertZDT0.toLocalDateTime();     // 1973-01-14T00:10
ZoneId theZID         = concertZDT0.getZone();             // US/Hawaii
ZoneOffset theZoffset = concertZDT0.getOffset();           // -10:00
```

```
int getFIELD()
```

Gets the value of the field designated by the suffix *`FIELD`*, which can be `DayOf-Month`, `DayOfYear`, `MonthValue`, `Nano`, `Second`, `Minute`, `Hour`, or `Year`.

Click here to view code image

```
DayOfWeek getDayOfWeek()
Month     getMonth()
```

Get the value of the day-of-week and month-of-year field, respectively.

```
ZoneId     getZone()
```

```
ZoneOffset getOffset()
```

Gets the time zone ID (e.g., `US/Central`) and the time zone offset from UTC/ Greenwich (e.g., `-04:00`), respectively (p. 1073).

Click here to view code image

```
int  get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

The first two methods return the value of the specified `TemporalField` (p. 1046) as an `int` value or as a `long` value, respectively. The value of the `ChronoField` enum constants `NANO_OF_DAY`, `MICRO_OF_DAY`, `EPOCH_DAY`, `PROLEPTIC_MONTH`, and `INSTANT_SECONDS` will not fit into an `int`, and therefore, the `getLong()` method must be used.

The third method checks if the specified field is supported by this zoned date-time. All `ChronoField` enum constants are supported by the `ZonedDateTime` class (p. 1046).

Click here to view code image

```
LocalTime     toLocalTime()
LocalDate     toLocalDate()
LocalDateTime toLocalDateTime()
```

Return the respective part of this zoned date-time.

```
Instant toInstant()
```

Converts a zoned date-time to an instant representing the same point as this date-time. This method is inherited by the `ZonedDateTime` class from its super-interface `java.time.chrono.ChronoZonedDateTime`.

Conversion of a zoned date-time to an instance of the `java.util.Date` legacy class can be done via this method (p. 1088).

Creating Modified Copies of Zoned Date-Time

Individual fields of a zoned date-time can be set to a new value in a copy of a zoned date-time as illustrated by the following code.

Click here to view code image

```
ZonedDateTime theZDT = concertZDT0          // 1973-01-14T00:10-10:00[US/Hawaii
    .withYear(1977)                         // 1977-01-14T00:10-10:00[US/Hawaii
    .with(ChronoField.MONTH_OF_YEAR, 8)     // 1977-08-14T00:10-10:00[US/Hawaii
    .withDayOfMonth(16)                     // 1977-08-16T00:10-10:00[US/Hawaii
    .with(ChronoField.HOUR_OF_DAY, 9)      // 1977-08-16T09:10-10:00[US/Hawaii
    .with(ChronoField.MINUTE_OF_HOUR, 30); // 1977-08-16T09:30-10:00[US/Hawaii
```

The `with`*`Field`*`()` methods behave analogous to the `with()` method with a corresponding field argument, taking into consideration the time gap and the time overlap that occur when daylight saving time starts and ends (p. 1082).

The `withZoneSameLocal()` method can be used to change the time zone, while retaining the date-time. The code at (1) below changes the time zone from `US/Hawaii` to `US/Central`, while retaining the date-time. Note that the two zoned date-times, `theZDT` and `zdtSameLocal`, do *not* represent the same instant according to UTC/Greenwich, as shown by the output from the print statements.

Click here to view code image

```
ZoneId cTZ = ZoneId.of("US/Central");
ZonedDateTime zdtSameLocal = theZDT.withZoneSameLocal(cTZ);            // (1)
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameLocal, zdtSameLocal.toInstant());
```

Output from the print statements:

Click here to view code image

```
          ZonedDateTime                   Instant
1977-08-16T09:30-10:00[US/Hawaii]   1977-08-16T19:30:00Z
1977-08-16T09:30-05:00[US/Central]  1977-08-16T14:30:00Z
```

The local time `09:30` in the `US/Hawaii` time zone is 10 hours (offset `-10:00`) behind UTC/Greenwich, whereas the same local time in the `US/Central` time zone (offset `-05:00`) is 5 hours behind UTC/Greenwich on the date `1977-08-16`, resulting in different instants at UTC/Greenwich for the same local time.

The `withZoneSameInstant()` method can be used to change the time zone *and* adjust the date-time to the new time zone. We see in the code below at (2) that the adjusted local time `14:30` in the `US/Central` time zone is 5 hours behind UTC/Greenwich, resulting in both zoned date-times representing the same instant at UTC/Greenwich.

Click here to view code image

```
ZonedDateTime zdtSameInstant = theZDT.withZoneSameInstant(cTZ);        // (2)
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameInstant, zdtSameInstant.toInstant());
```

Output from the print statements:

Click here to view code image

```
           ZonedDateTime                      Instant
1977-08-16T09:30-10:00[US/Hawaii]   1977-08-16T19:30:00Z
1977-08-16T14:30-05:00[US/Central]  1977-08-16T19:30:00Z
```

Click here to view code image

```
ZonedDateTime withFIELD(int amount)
```

Returns a copy of this zoned date-time with the field designated by the suffix *`FIELD`* set to the specified value, where the suffix *`FIELD`* can be `DayOfMonth`, `DayOfYear`, `MonthValue`, `Nano`, `Second`, `Minute`, `Hour`, or `Year`.

Note that these methods take into consideration the time gap and the time overlap that can occur due to daylight savings (p. 1082).

Click here to view code image

```
ZonedDateTime with(TemporalField field, long newValue)
```

Returns a copy of this zoned date-time with the specified `TemporalField` set to the specified value. All constants of the `ChronoField` enum type (p. 1046) can be used to specify a particular field.

Click here to view code image

```
ZonedDateTime withZoneSameLocal(ZoneId zone)
ZonedDateTime withZoneSameInstant(ZoneId zone)
```

The first method returns a copy of this zoned date-time with the specified time zone, but normally retaining the date-time.

The second method returns a copy of this zoned date-time with the specified time zone, but retaining the instant; that is, it results in the date-time being adjusted according to the specified time zone.

Temporal Arithmetic with Zoned Date-Time

Temporal arithmetic with zoned date-times is analogous to temporal arithmetic with date-times (p. 1040). However, it is important to note the differences that are due to time zones and daylight savings (p. 1082).

Example 17.8 illustrates calculating flight times. The code at (1) creates a flight departure time for a flight to London from New York at 8:30 pm on July 4, 2021. The flight time is 7 hours and 30 minutes. The code at (2) calculates the local time of arrival at London, first by calling the `withZoneSameInstant()` method to find the local time in London at the time the flight departs from New York, and then adding the flight time to obtain the local arrival time in London.

In Example 17.8, the code at (3) calculates the flight duration by calling the `Duration.between()` method with the departure and arrival times as arguments. The code at (4) calls the `until()` method to calculate the flight time in minutes from the departure time to the arrival time.

Finally, the code at (5) and (6) calculates the local time at the departure airport at the time the flight arrives in London in two different ways. At (5), the `plusMinutes()` method adds the flight time to the departure time. At (6), the `withZoneSameInstant()` method converts the arrival time to its equivalent in the departure time zone.

Example 17.8 *Flight Time Information*

Click here to view code image

```
import java.time.DateTimeException;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;

public class FlightTimeInfo {
  public static void main(String[] args) {
    try {
      // Departure from New York at 8:30pm on July 4, 2021.                (
      LocalDateTime departure = LocalDateTime.of(2021, Month.JULY, 4, 20, 30);
      ZoneId departureZone = ZoneId.of("America/New_York");
      ZonedDateTime departureZDT = ZonedDateTime.of(departure, departureZone);

      // Flight time is 7 hours and 30 minutes.
      // Calculate local arrival time at London:                          (
      ZoneId arrivalZone = ZoneId.of("Europe/London");
      ZonedDateTime arrivalZDT
          = departureZDT.withZoneSameInstant(arrivalZone)
                        .plusMinutes(7*60 + 30);
      System.out.printf("DEPARTURE:  %s%n", departureZDT);
      System.out.printf("ARRIVAL:    %s%n", arrivalZDT);

      // Flight time as a Duration:                                        (
      Duration flightduration = Duration.between(departureZDT, arrivalZDT);
      System.out.println("Flight duration:     " + flightduration);
```

```
     // Flight time in minutes:                                         (
     long flightTime = departureZDT.until(arrivalZDT, ChronoUnit.MINUTES);
     System.out.println("Flight time (mins.): " + flightTime);

     System.out.printf(                                               // (
       "Time at departure airport on arrival: %s%n",
       departureZDT.plusMinutes(7*60 + 30));

     System.out.printf(                                               // (
       "Time at departure airport on arrival: %s%n",
       arrivalZDT.withZoneSameInstant(departureZone));

    } catch (DateTimeException e) {
     e.printStackTrace();
    }
   }
  }
```

Output from the program:

Click here to view code image

```
DEPARTURE:  2021-07-04T20:30-04:00[America/New_York]
ARRIVAL:    2021-07-05T09:00+01:00[Europe/London]
Flight duration:     PT7H30M
Flight time (mins.): 450
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
```

Selected methods from the `ZonedDateTime` class for temporal arithmetic are presented below.

> Click here to view code image
>
> ```
> ZonedDateTime plusUNIT(long amount)
> ZonedDateTime minusUNIT(long amount)
> ```
>
> Return a copy of this zoned date-time with the specified `amount` either added or subtracted, respectively, where the unit is designated by the suffix *`UNIT`*, which can be `Nanos`, `Seconds`, `Minutes`, `Hours`, `Days`, `Weeks`, `Months`, or `Years`.
>
> With the time units (`Nanos`, `Seconds`, `Minutes`, `Hours`), the operation is on the instant timeline, where a `Duration` of the specified amount is added or subtracted.
>
> With the date units (`Days`, `Weeks`, `Months`, `Years`), the operation is on the local timeline, adding and subtracting the amount from the date-time corresponding to this zoned date-time.
>
> Click here to view code image
>
> ```
> ZonedDateTime plus(long amountToAdd, TemporalUnit unit)
> ZonedDateTime minus(long amountToSubtract, TemporalUnit unit)
> boolean isSupported(TemporalUnit unit)
> ```

Return a copy of this zoned date-time with the specified amount added or subtracted, respectively, according to the `TemporalUnit` specified. The methods support all `ChronoUnit` enum constants, except `FORVER` (p. 1044).

Time units operate on the instant timeline, but date units operate on the local timeline, when performing these operations.

The `isSupported()` method checks if the specified `TemporalUnit` is supported by this zoned date-time (p. 1044).

Click here to view code image

```
ZonedDateTime plus(TemporalAmount amountToAdd)
ZonedDateTime minus(TemporalAmount amountToSubtract)
```

Return a copy of this zoned date-time with the specified temporal amount added or subtracted, respectively. The classes `Period` (p. 1057) and `Duration` (p. 1064) implement the `TemporalAmount` interface.

If the amount is a `Duration`, the time units operate on the instant timeline. However, if the amount is a `Period`, the date units operate on the local timeline.

Click here to view code image

```
long until(Temporal endExclusive, TemporalUnit unit)
```

Calculates the amount of time between this zoned date-time and the specified temporal object in terms of the specified `TemporalUnit` (p. 1044). The end point is excluded. An exception is thrown if the specified temporal object cannot be converted to a zoned date-time.

This method supports all `ChronoUnit` enum constants, except `FORVER` (p. 1044).

Date units operate on the local timeline, but time units operate on the instant timeline when performing this operation.

Daylight Savings

If a time zone practices daylight savings, the time zone offset of a `ZonedDateTime` can be different depending on whether daylight saving time (DST) or standard time (for the rest of the year) is in effect. Zone rules associated with the zone ID of a zoned date-time are used to determine the right zone offset for time zones that practice daylight savings.

A *time gap* occurs when the clock is moved *forward* at the start of DST, and a *time overlap* occurs when the clock is moved *backward* at the end of DST. Usually the gap and the overlap are 1 hour. In that case, when the time gap occurs, an hour is lost and that day has only 23 hours, whereas when a time overlap occurs, an hour is gained and that day has 25 hours. Care should be exercised when dealing with zoned date-times that can involve crossovers to and from DST. Luckily, the methods `of()`, `with()`, `plus()`, and `minus()` of the `ZonedDateTime` class take daylight savings into consideration.

As an example to illustrate daylight savings, we will use the `US/Central` time zone. For this time zone, DST starts at 02:00:00 on 14 March in 2021, when the clocks are moved forward by 1 hour, resulting in the hour between 02:00:00 and 03:00:00 being lost. The gap in this case is 1 hour. The day on 2021-03-14 will only be 23 hours long.

DST for the `US/Central` time zone ends at 02:00:00 on 7 November in 2021, when clocks are moved backward by 1 hour, resulting in the hour before 02:00:00 being repeated twice. The overlap in this case is 1 hour. The day on 2021-11-07 will be 25 hours long.

The following output from Example 17.9 illustrates what happens when the `plus-Hours()` method increments a zoned date-time across the time gap.

Click here to view code image

```
  Daylight Savings in US/Central starts at 2021-03-14T02:00 (spring forward 1 ho
                      ___________ZonedDateTime__________
                         Date    Time  Offset    TZ       DST   UTC
  (1) Before gap:     2021-03-14 01:30 -06:00 US/Central false 07:30
      + 1 hour
  (2) After gap:      2021-03-14 03:30 -05:00 US/Central true  08:30
```

Line (1) above shows the following information about a zoned date-time that is before the gap: `Date` (`2021-03-14`), `Time` (`01:30`), `Offset` (`-06:00`), `TZ` (`US/Central`), `DST` that indicates whether it is in effect (`false`), and `UTC` equivalent of the time (`07:30`). Note that the time `01:30` is before the gap. Adding 1 hour to the zoned date-time in line (1) puts the resulting time (`02:30`) in the gap, which does not exist. The `plusH-ours()` method increments the expected time `02:30` by the gap length to `03:30` and increments the offset `-6:00` by the gap length to `-05:00` to comply with DST. Line (2) shows the final result of the operation. DST is now in effect. The UTC equivalent time is now `08:30`, an hour after `07:30`, as we would expect.

The following output from Example 17.9 illustrates what happens when the `plus-Hours()` method increments a zoned date-time successively across the time overlap.

Click here to view code image

```
Daylight Savings in US/Central ends at 2021-11-07T02:00 (fall back 1 hour).
                    ___________ZonedDateTime__________
                       Date    Time  Offset    TZ         DST   UTC
(1) Before overlap: 2021-11-07 00:30 -05:00 US/Central true  05:30
    + 1 hour
(2) In overlap:     2021-11-07 01:30 -05:00 US/Central true  06:30
    + 1 hour
(3) In overlap:     2021-11-07 01:30 -06:00 US/Central false 07:30
    + 1 hour
(4) After overlap:  2021-11-07 02:30 -06:00 US/Central false 08:30
```

Line (1) shows a zoned date-time before the overlap with the following information: `Date` (`2021-11-07`), `Time` (`00:30`), `Offset` (`-05:00`), `TZ` (`US/Central`), `DST` that indicates whether it is in effect (`true`), and `UTC` equivalent of the time (`05:30`). Again note that the time `00:30` is before the time overlap. Lines (2), (3), and (4) show the result of successively adding 1 hour to the resulting zoned date-time from the previous operation, starting with the zoned date-time in line (1).

Adding 1 hour to the time `00:30` in line (1) with DST in effect changes the time to `01:30`, which is in the overlap, but before the DST crossover at `02:00`. The result is shown in line (2).

Adding 1 hour to the time `01:30` in line (2) with DST in effect does *not* change the time. The operation only decrements the offset `-05:00` by the overlap length (1 hour) to `-06:00`, as the time `01:30` is still in the overlap after the DST crossover because it is repeated, but now the standard time is in effect. The result is shown in line (3).

Adding 1 hour to the time `01:30` in line (3) with standard time in effect changes the time to `02:30`. The resulting time `02:30` is not in the overlap. The result is shown in line (4). The result would have been the same if we had added 3 hours to the zoned date-time in line (1): the time would be incremented by 2 hours (3 – overlap length) and the offset decremented by the overlap length (1 hour).

The last column, `UTC`, also shows that the time at UTC/Greenwich changed successively by 1 hour as a result of the plus operations.

In Example 17.9, the methods `adjustForGap()` at (1a) and `adjustForOverlap()` at (1b) create the scenarios for DST crossovers discussed above. The method `printInfo()` at (7) prints the result of each plus operation. The essential lines of code are (4a) and (4b) that perform the plus operations, with the rest of the code creating zoned date-times (2a, 3a, 2b, 3b) and printing formatted output (5a, 5b).

The method `isDST()` at (8) determines if DST is in effect for a zoned date-time. The method `localTimeAtUTC()` at (9) returns the UTC equivalent of the time in a zoned date-time. These auxiliary methods are simple but instructive examples of operations on zoned date-times.

The zone rules associated with a time zone can be obtained by calling the `ZoneId.getRules()` method on a zone ID. The zone rules are represented by the `java.time.zone.ZoneRules` class that provides the `isDaylightSavings()` method to determine whether an instant is in daylight savings. A zoned date-time can be converted to an instant by the `toInstant()` method.

> Click here to view code image
>
> ```
> // java.time.zone.ZoneRules
> boolean isDaylightSavings(Instant instant)
> ```
>
> Determines whether the specified `instant` is in daylight savings.

Example 17.9 *Adjusting for DST Crossovers*

Click here to view code image

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;

public class DSTAdjustment {
  public static void main(String[] args) {
    adjustForGap();
    adjustForOverlap();
  }

  /**
   * Adjustment due to the time gap at DST crossover.                       (1
   * DST starts in US/Central TZ: 2021-03-14T02:00:00,
   * clocks are moved forward 1 hour, resulting in a time gap of 1 hour.
   */
  static void adjustForGap() {
    // Start date and time for DST in US/Central in 2021.                   (2
    ZoneId cTZ = ZoneId.of("US/Central");
    LocalDate dateStartDST = LocalDate.of(2021, 3, 14);
    LocalTime timeStartDST = LocalTime.of(2, 0);
```

```
    LocalDateTime ldtStartDST = LocalDateTime.of(dateStartDST, timeStartDST);
    ZonedDateTime zdtStartDST = ZonedDateTime.of(ldtStartDST, cTZ);

    // Time before the gap.                                                  (3
    LocalTime timeBeforeGap = LocalTime.of(1, 30);
    LocalDateTime ldtBeforeGap = LocalDateTime.of(dateStartDST, timeBeforeGap)
    ZonedDateTime zdtBeforeGap = ZonedDateTime.of(ldtBeforeGap, cTZ);

    // Add 1 hour.                                                           (4
    ZonedDateTime zdtAfterGap = zdtBeforeGap.plusHours(1);

    // Print a report.                                                       (5
    System.out.printf("Daylight Savings in %s starts at %s "
        + "(spring forward 1 hour).%n", cTZ, ldtStartDST);
    System.out.println("                    ___________ZonedDateTime__________

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before gap:     ", zdtBeforeGap);
    System.out.println("    + 1 hour");
    printInfo("(2) After gap:      ", zdtAfterGap);
    System.out.println();

    // Add 3 hours:                                                          (6
    ZonedDateTime zdtPlus3Hrs = zdtBeforeGap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeGap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Adjustment due to the time overlap at DST crossover.                      (
   * DST ends in US/Central TZ: 2021-11-07T02:00:00,
   * clocks are moved backward 1 hour, resulting in a time overlap of 1 hour.
   */
  static void adjustForOverlap() {
    // End date and time for DST in US/Central in 2021.                      (2
    ZoneId cTZ = ZoneId.of("US/Central");
    LocalDate dateEndDST = LocalDate.of(2021, 11, 7);
    LocalTime timeEndDST = LocalTime.of(2, 0);
    LocalDateTime ldtEndDST = LocalDateTime.of(dateEndDST, timeEndDST);
    ZonedDateTime zdtEndDST = ZonedDateTime.of(ldtEndDST, cTZ);

    // Time before the overlap:                                              (3
    LocalTime timeBeforeOverlap = LocalTime.of(0, 30);
    LocalDateTime ldtBeforeOverlap = LocalDateTime.of(dateEndDST,
                                                       timeBeforeOverlap);
    ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(ldtBeforeOverlap, cTZ);

    // Add 1 hour:                                                           (4
    ZonedDateTime zdtInOverlap1 = zdtBeforeOverlap.plusHours(1);
    ZonedDateTime zdtInOverlap2 = zdtInOverlap1.plusHours(1);
    ZonedDateTime zdtAfterOverlap = zdtInOverlap2.plusHours(1);
    // Print a report.                                                       (5
    System.out.printf("Daylight Savings in %s ends at %s (fall back 1 hour).%n
                      cTZ, ldtEndDST);
    System.out.println("                    ___________ZonedDateTime__________

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before overlap: ", zdtBeforeOverlap);
    System.out.println("    + 1 hour");
    printInfo("(2) In overlap:     ", zdtInOverlap1);
    System.out.println("    + 1 hour");
    printInfo("(3) In overlap:     ", zdtInOverlap2);
    System.out.println("    + 1 hour");
    printInfo("(4) After overlap:  ", zdtAfterOverlap);
    System.out.println();

    // Add 3 hours:                                                         (6
    ZonedDateTime zdtPlus3Hrs = zdtBeforeOverlap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeOverlap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Print info for a date-time.                                            (7
   * @param leadTxt Text to lead the information.
   * @param zdt     Zoned date-time whose info is printed.
   */
  static void printInfo(String leadTxt, ZonedDateTime zdt) {
    System.out.printf(leadTxt + "%10s %5s %6s %5s %-5s %5s%n",
                      zdt.toLocalDate(), zdt.toLocalTime(),
                      zdt.getOffset(), zdt.getZone(),
                      isDST(zdt), localTimeAtUTC(zdt));
  }

  /**
   * Determine if DST is in effect for a zoned date-time.                   (8
   * @param zdt  Zoned date-time whose DST status should be determined.
   * @return     true, if DST is in effect.
   */
  static boolean isDST(ZonedDateTime zdt) {
    return zdt.getZone().getRules().isDaylightSavings(zdt.toInstant());
  }

  /**
   *  Find local time at UTC/Greenwich equivalent to local time in          (9
   *  the specified zoned date-time.
   * @param zdt   Zoned date-time to convert to UTC/Greenwich.
   * @return      Equivalent local time at UTC/Greenwich.
   */
  static LocalTime localTimeAtUTC(ZonedDateTime zdt) {
    return zdt.withZoneSameInstant(ZoneOffset.UTC).toLocalTime();
  }
}
```

Analogous to the `plus()` methods, the `of()` methods also make adjustments at DST crossings. Sticking to the DST information from Example 17.9, if we try to create a zoned date-time with a time that is in the gap, the `of()` method typically moves the time by the length of the gap (1 hour) into DST.

Click here to view code image

```
ZonedDateTime zdt1 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14),
    LocalTime.of(2, 30),                // Time 02:30 is in the gap.
    ZoneId.of("US/Central"));
System.out.println(zdt1);               // 2021-03-14T03:30-05:00[US/Central]
```

For a time in the overlap, the offset is ambiguous—it can be -05:00 for DST or -06:00 for standard time. Typically, the `of()` methods return a zoned date-time with the DST offset, as shown in the following code:

Click here to view code image

```
ZonedDateTime zdt2 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(1, 30),                // Time 01:30 in the overlap.
    ZoneId.of("US/Central"));
System.out.println(zdt2);               // 2021-11-07T01:30-05:00[US/Central]
```

The `with`*`Field`*`()` methods behave analogously to the `to()` methods when it comes to DST crossings. If we try to create a zoned date-time with the time 02:00 that is in the gap, the `withHour()` method typically moves the time by the length of the gap (1 hour) into DST.

Click here to view code image

```
ZonedDateTime zdt3 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14), LocalTime.of(0, 0), ZoneId.of("US/Central")
  ).withHour(2);                     // Time 02:00 is in the gap.
System.out.println(zdt3);            // 2021-03-14T03:00-05:00[US/Central]
```

For a time in the overlap, the `withMinute()` method returns a zoned date-time with the DST offset, as shown in the following code:

Click here to view code image

```
ZonedDateTime zdt4 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7), LocalTime.of(1, 0), ZoneId.of("US/Central")
  ).withMinute(30);                 // Time 01:30 is in the overlap.
System.out.println(zdt4);           // 2021-11-07T01:30-05:00[US/Central]
```

Finally, the result of temporal arithmetic with zoned date-times can depend on whether date units or time units are involved. We illustrate the behavior using the following zoned date-time that is before the crossover from DST to standard time.

Click here to view code image

```
ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(0, 30),             // Time 00:30 is before the DST crossover
    ZoneId.of("US/Central"));
```

We add 1 day with the `plusDays()` method and 24 hours with the `plusHours`() method to the zoned date-time using the code below. Number of days is measured by the date unit *days*, and number of hours by the time unit *hours*. From the output we see that the results are different. The day was added to the date-time and converted back to a zoned date-time with the offset adjusted, without affecting the time part; that is, date units operate on the *local* timeline. Adding 24 hours results in a zoned-based time that is exactly a duration of 24 hours later, taking the DST crossover into consideration; that is, time units operate on the *instant* timeline.

Click here to view code image

```
// Date units and time units.
System.out.printf("%s + 1 day    = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusDays(1));        // (1) Add 1 day.
System.out.printf("%s + 24 hours = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusHours(24));      // (2) Add 24 hour
```

Output from the print statements:

Click here to view code image

```
2021-11-07T00:30-05:00[US/Central] + 1 day    = 2019-11-04T00:30-06:00[US/Cent
2021-11-07T00:30-05:00[US/Central] + 24 hours = 2021-11-07T23:30-06:00[US/Cent
```

By the same token, adding a `Period` (that has only date fields) and a `Duration` (that has only time fields) using the `plus(Period.ofDays(1))` and the `plus`(Duration .ofHours(24)) method calls instead at (1) and (2) above, respectively, will give the same results.

17.8 Converting Date and Time Values to Legacy Date

An object of the `java.util.Date` legacy class represents time, date, and time zone. The class provides the method `from()` to convert a `java.time.Instant` (p. 1049) to a `Date`. In order to convert dates and times created using the new Date and Time API, we need to go through an `Instant` to convert them to a `Date` object.

The `java.time.ZonedDateTime` class provides the `toInstant()` method to convert a `ZonedDateTime` object to an `Instant`, which is utilized by the `zdtToDate()` method of the `ConvertToLegacyDate` utility class in Example 17.10 at (1).

A `java.time.LocalDateTime` object lacks a time zone in order to convert it to a `Date`. The `ldtToDate()` method at (2) adds the system default time zone to create a `Zoned-DateTime` object which is then converted to an `Instant`.

A `LocalDate` object lacks time and a time zone in order to convert it to a `Date`. The `ldToDate()` method at (3) adds a fictive time (start of the day), and the resulting `LocalDateTime` object is added the system default time zone to create a `ZonedDateTime` object which is then converted to an `Instant`.

A `LocalTime` object lacks a date and a time zone in order to convert it to a `Date`. The `ltToDate()` method at (4) adds a fictive date (2021-1-1), and the resulting `LocalDateTime` object is added to the system default time zone to create a `ZonedDateTime` object which is then converted to an `Instant`.

For an example, see Example 18.8, p. 1143.

Example 17.10 *Converting to Legacy Date*

Click here to view code image

```
import java.time.*;
import java.util.Date;

public class ConvertToLegacyDate {
  /** Convert a ZonedDateTime to Date. */
  public static Date zdtToDate(ZonedDateTime zdt) {                         //
    return Date.from(zdt.toInstant());
  }

  /** Convert a LocalDateTime to Date. */
  public static Date ldtToDate(LocalDateTime ldt) {                         //
    return Date.from(ldt.atZone(ZoneId.systemDefault()).toInstant());
  }

  /** Convert a LocalDate to Date. */
  public static Date ldToDate(LocalDate ld) {                               //
    return Date.from(ld.atStartOfDay()
                        .atZone(ZoneId.systemDefault())
                        .toInstant());
  }

  /** Convert a LocalTime to Date. */
  public static Date ltToDate(LocalTime lt) {                               //
    return Date.from(lt.atDate(LocalDate.of(2021, 1, 1))
                        .atZone(ZoneId.systemDefault())
                        .toInstant());
  }
}
```

Review Questions

17.1 Given the following code:

Click here to view code image

```
import java.time.LocalDate;
public class RQ1 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 31);
    LocalDate d2 = d1.plusMonths(1);
    LocalDate d3 = d2.minusMonths(1);
    System.out.println(d1.getDayOfYear() + " " + d2.getDayOfYear() + " "
                        d3.getDayOfYear());
    }
  }
```

What is the result?

Select the one correct answer.

a. `31 61 31`

b. `31 59 28`

c. `31 59 31`

d. The program will throw an exception at runtime.

17.2 Given the following code:

Click here to view code image

```
import java.time.LocalDate;
public class RQ2 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 1);
    d1 = d1.withDayOfMonth(d1.lengthOfMonth()).withMonth(2);
    System.out.println(d1);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-02-28`

b. `2021-02-31`

c. `2021-03-03`

d. The program will throw an exception at runtime.

17.3 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ3 {
  public static void main(String[] args) {
    LocalDateTime d1 = LocalDate.of(2021, 4, 1).atStartOfDay();
    Instant i1 = d1.toInstant(ZoneOffset.of("+18:00"));
    LocalDate d2 = LocalDate.ofInstant(i1, ZoneId.of("UTC"));
    System.out.println(d2);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-1`

b. `2021-04-2`

c. `2021-03-30`

d. `2021-03-31`

17.4 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ4 {
  public static void main(String[] args) {
    LocalDateTime dt = LocalDate.of(2021, 4, 1).atStartOfDay();
    ZonedDateTime zdt1 = dt.atZone(ZoneId.of("Europe/Paris"));
    ZonedDateTime zdt2 = dt.atZone(ZoneId.of("Europe/London"));
    Duration d = Duration.between(zdt1.minusMinutes(30), zdt2.plusMinute
    System.out.println(d);
  }
}
```

What is the result, given that the time difference between Paris and London is 1 hour?

Select the one correct answer.

a. `PT0H`

b. `PT1H`

c. `PT2H`

d. `PT-2H`

e. `PT-1H`

17.5 Which statement is false?

Select the one correct answer.

a. `Instant` objects can represent points in time with nanosecond precision.

b. `Instant` objects and `LocalTime` objects have same precision.

c. `Duration` objects can express the amount of time between two `LocalDate` objects.

d. `Period` objects can express the number of days between two `Instant` objects.

17.6 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ6 {
  public static void main(String[] args) {
    LocalTime t = LocalTime.of(8, 15);
    LocalDate d = LocalDate.of(2021, 4, 1);
    LocalDateTime dt = d.atTime(t);
    dt.minusMinutes(30).withDayOfMonth(12);
    System.out.println(dt);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-12T08:45`

b. `2021-04-12T07:45`

c. `2021-04-01T08:15`

d. `2021-04-01T07:15`

17.7 Which statement is true?

Select the one correct answer.

a. Adding a `Duration` and adding a `Period` to a `ZonedDateTime` object produces the same result.

b. A `Duration` of 36,000 seconds is the same as a `Period` of 1 hour.

c. Daylight savings is taken into consideration by `LocalTime` and `LocalDateTime` objects.

d. A `Period` of 1 day is always equivalent to a `Duration` of 24 hours.

e. `Period` and `Duration` objects can have positive and negative values.

17.8 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ8 {
  public static void main(String[] args) {
    Period d  = Period.parse("P2D");
    LocalDate ld = LocalDate.of(2021, 4, 1);
    LocalDateTime ldt = ld.plus(d);
    System.out.println(ldt);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-04-01T00:30`

b. `2021-04-01T23:30`

c. `2021-03-31T00:30`

d. `2021-03-31T23:30`

e. The program will fail to compile.

17.9 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ9 {
  public static void main(String[] args) {
    Duration d  = Duration.parse("PT-24H");
    LocalDate ld = LocalDate.of(2021, 4, 1).plus(d);
    System.out.println(ld);
  }
}
```

What is the result?

Select the one correct answer.

a. `2021-03-31`

b. `2021-04-01`

c. `2021-04-02`

d. The program will throw an exception at runtime.

e. The program will fail to compile.

17.10 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ10 {
  public static void main(String[] args) {
    LocalDate ld = LocalDate.of(2021, 4, 1);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

Which of the following statements, when inserted independently at (1), will print the following result: `2021-04-03T00:30`?

Select the two correct answers.

a. Click here to view code image

```
LocalDateTime ldt = ld.atTime(LocalTime.of(0, 30))
                      .plus(Duration.ofHours(48));
```

b. Click here to view code image

```
LocalDateTime ldt = ld.plus(Duration.ofHours(48))
                      .atTime(LocalTime.of(0, 30));
```

c. Click here to view code image

```
LocalDateTime ldt = ld.atTime(LocalTime.of(48,30));
```

d. Click here to view code image

```
LocalDateTime ldt = ld.plusDays(3).atTime(LocalTime.of(-23,30));
```

e. Click here to view code image

```
LocalDateTime ldt = ld.atTime(0, 30).plus(Duration.ofHours(48));
```

17.11 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ11 {
  public static void main(String[] args) {
    LocalDate now = LocalDate.now();
    LocalDate foolsDay = LocalDate.of(2021, Month.APRIL, 1);
    LocalDateTime tomorrowAfternoon = now.plusDays(1)
                                         .atTime(LocalTime.of(12, 01));
    LocalDate anotherDay = foolsDay.withDayOfMonth(2).minusDays(1);
  }
}
```

How many `LocalDate` objects are created in this example?
Select the one correct answer.

a. `3`
b. `4`
c. `5`
d. `6`

17.12 Given the following code:

Click here to view code image

```
import java.time.*;
public class RQ13 {
  public static void main(String[] args) {
    LocalTime lt = LocalTime.of(17,30);
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    Duration d = Duration.between(lt, ldt);
    System.out.println(d);
  }
}
```

What is the result?
Select the one correct answer.

a. `PT-1H-45M`
b. `PT1H-45M`
c. `PT2H-15M`
d. `PT-2H-15M`
e. The program will throw an exception at runtime.

17.13 Given the following code:

Click here to view code image

```
import java.time.*;
import static java.time.temporal.ChronoUnit.DAYS;
public class RQ14 {
  public static void main(String[] args) {
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

Which statement inserted at (1) will *not* give the following result: `2021-04-03T16:15`?
Select the one correct answer.

a. Click here to view code image

```
ldt = ldt.plusHours(1).with(LocalDate.of(2021, Month.APRIL, 3));
```

b. Click here to view code image

```
ldt = ldt.plusDays(1).with(LocalTime.of(16, 15));
```

c. Click here to view code image

```
ldt = ldt.plus(Duration.of(2, DAYS)).minus(Duration.parse("PT23H"));
```

d. Click here to view code image

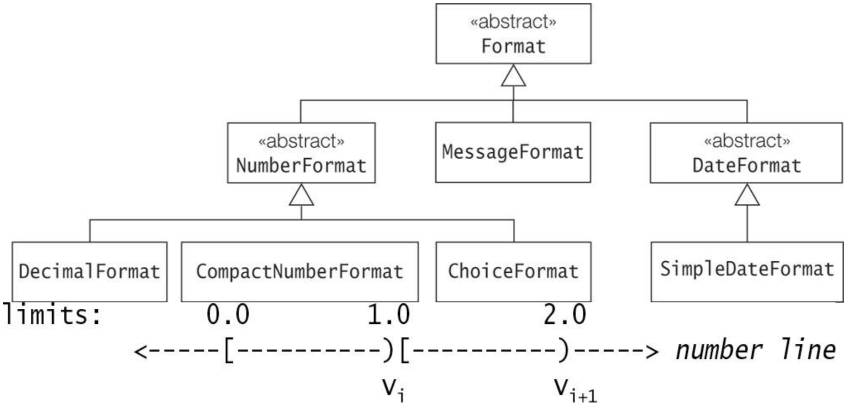
```
ldt = ldt.plus(Duration.of(2, DAYS))
         .minus(Duration.ofMinutes(15).ofHours(16));
```

e. Click here to view code image

```
ldt = ldt.plus(Duration.parse("PT25H"));
```

f. Click here to view code image

```
ldt = ldt.plus(Duration.ofMinutes(25 * 60));
```


«abstract»
Format
«abstract»
NumberFormat
MessageFormat
«abstract»
DateFormat
DecimalFormat
CompactNumberFormat
ChoiceFormat
SimpleDateFormat
limits: 0.0 1.0 2.0
<-----[----------)[----------)-----> number line
V_i V_{i+1}

Localization 18

Chapter Topics

- Understanding localization of applications
- Understanding the role and purpose of locales as embodied by the `java.util.Locale` class
- Understanding how to specify a properties file
- Creating and using resource bundles with the `java.util.Resource-Bundle` class for storing and retrieving locale-specific information for the purpose of localization
- Using formatters provided by the `java.text.NumberFormat` class and its subclass, `java.text.DecimalFormat`, for locale-specific formatting and parsing of number, currency, and percentage values, including compact number and

accounting currency formatting

- Using formatters provided by the `java.time.format.DateTimeFormatter` class for locale-specific formatting and parsing of date and time values
- Using formatters provided by the `java.text.MessageFormat` class for locale-specific formatting and parsing of messages
- Using the `java.text.ChoiceFormat` class for implementing choice patterns for conditional formatting and parsing of messages

Java SE 17 Developer Exam O	
[18.1] Implement localization using locales, resource bundles, parse and format messages, dates, times, and numbers including currency and percentage values	*§18.1, p. 1096* *to* *§18.7, p. 1139*
Java SE 11 Developer Exam O	
[12.1] Implement Localization using Locale, resource bundles, and Java APIs to parse and format messages, dates, and numbers	*§18.1, p. 1096* *to* *§18.7, p. 1139*

Often, applications have to work across borders and cultures, and thus must be capable of adjusting to a variety of local requirements. For example, an accounting system for the US market is obviously not going to function well in the Norwegian market. Not only is the formatting of dates, times, numbers, and currency different, but the languages in the two markets are different as well. Developing programs so that they have global awareness of such differences is called *internationalization*—also known as *i18n* (the 18 refers to the 18 characters deleted in the word *internationalization*).

Java provides the concept of a *locale* to create applications that adhere to cultural and regional preferences necessary in order to make an application truly international—a process called *localization*. Localization of applications enhances the user experience by being culturally sensitive to the user and minimizing misinterpretation of information on the part of the user. The user dialogue is an obvious candidate for localization in an application. Identifying and factorizing to isolate locale-sensitive data and code facilitates localization. Understanding cultural issues is important in this regard. Other examples of resources that might require localization are messages, salutations, graphics, measurement systems, postal codes, and phone numbers. Hard-coding such locale-sensitive information is certainly not a good idea. Once an application is properly set up for internationalization, it is relatively straightforward to localize it for multiple locales without too much effort.

This chapter covers support provided by the Java SE Platform APIs to aid localization of applications: locales to access particular cultural and regional preferences, properties files as resource bundles to store locale-specific information, and formatters to format numbers, currencies, dates, times, and messages according to locale rules.

18.1 Using Locales

A *locale* represents a specific geographical, political, or cultural region. Its two most important attributes are *language* and *country*. Certain classes in the Java standard library provide *locale-sensitive* operations. For example, they provide methods to format values that represent *dates*, *currencies*, and *numbers* according to a specific locale. Adapting a program to a specific locale is called *localization*.

A locale is represented by an instance of the class `java.util.Locale`. A locale object can be created using the following constructors:

> Click here to view code image
>
> ```
> Locale(String language)
> Locale(String language, String country)
> ```
>
> The `language` string is normalized to lowercase and the `country` string to upper-case. The language argument is an ISO-639 Language Code (which uses two lowercase letters), and the country argument is an ISO-3166 Country Code (which uses two uppercase letters) or a UN M.49 three-digit area code— although the constructors do not impose any constraint on their length or perform any syntactic checks on the arguments.
>
> For the one-argument constructor, the country remains undefined.

Examples of selected language codes and country codes are given in Table 18.1 and Table 18.2, respectively.

Table 18.1 *Selected Language Codes*

Language code

`"en"`	English
`"no"`	Norwegian
`"fr"`	French

Table 18.2 *Selected Country/Region Codes*

Country/Region code	Country/Region
`"US"`	United States (U.S.)
`"GB"`	Great Britain (GB)
`"NO"`	Norway
`"FR"`	France
`"003"`	North America

The `Locale` class also has predefined locales for certain *languages*, irrespective of the region where they are spoken, as shown in Table 18.3.

Table 18.3 *Selected Predefined Locales for Languages*

Constant	Language
`Locale.ENGLISH`	Locale with English (`new Locale("en")`)
`Locale.FRENCH`	Locale with French (`new Locale("fr")`)
`Locale.GERMAN`	Locale with German (`new Locale("de")`)—that is, Deutsch

The `Locale` class also has predefined locales for certain *combinations of countries and languages*, as shown in Table 18.4.

Table 18.4 *Selected Predefined Locales for Countries*

Constant	Country
`Locale.US`	Locale for U.S. (`new Locale("en","US")`)
`Locale.UK`	Locale for United Kingdom/Great Britain (`new Locale("en","GB")`)
`Locale.CANADA_FRENCH`	Locale for Canada with French language (`new Locale("fr","CA")`)

Normally a program uses the *default locale* on the platform to provide localization. The `Locale` class provides a get and a set method to manipulate the default locale.

> Click here to view code image
>
> ```
> static Locale getDefault()
> static void   setDefault(Locale newLocale)
> ```
>
> The first method returns the default locale, and the second one sets a specific locale as the default locale.
>
> Click here to view code image
>
> ```
> static Locale[] getAvailableLocales()
> ```

Returns an array of all installed locales.

Click here to view code image

```
static Locale forLanguageTag(String languageTag)
```

Returns a locale for the specified IETF BCP 47 language tag string, which allows extended locale properties, such as a calendar or a numeric system. An example would be creating a locale based on the language tag `"zh-cmn-Hans-CN"`, which stands for Mandarin Chinese, Simplified script, as used in China. Note that the *locale qualifiers* (also called *subtags*) in the language tag are separated by hyphens (-).

```
String getCountry()
```

Returns the country/region code for this locale, or `null` if the code is not defined for this locale.

```
String getLanguage()
```

Returns the language code of this locale, or `null` if the code is not defined for this locale.

Click here to view code image

```
String getDisplayCountry()
String getDisplayCountry(Locale inLocale)
```

Returns a name for the locale's country that is appropriate for display, depending on the default locale in the first method or the `inLocale` argument in the second method.

Click here to view code image

```
String getDisplayLanguage()
String getDisplayLanguage(Locale inLocale)
```

Return a name for the locale's language that is appropriate for display, depending on the default locale in the first method or the `inLocale` argument in the second method.

Click here to view code image

```
String getDisplayName()
String getDisplayName(Locale inLocale)
```

Return a name for the locale that is appropriate for display.

```
String toString()
```

Returns a text representation of this locale in the format `"`*`languageCode_countryCode`*`"`. Language is always lowercase and country is always uppercase. If either of the codes is not

> specified in the locale, the _ (underscore) is omitted.

A locale is an immutable object, having *two sets* of get methods to return the *display name* of the country and the language in the locale. The first set returns the display name of the current locale according to the default locale, while the second set returns the display name of the current locale according to the locale specified as an argument in the method call.

Methods that require a locale for their operation are called *locale-sensitive*. Methods for formatting numbers, currencies, dates, and the like use the locale information to determine how such values should be formatted. A locale does not provide such services. Subsequent sections in this chapter provide examples of locale-sensitive classes (Figure 18.1, p. 1115).

Figure 18.1 *Core Classes for Formatting in the* `java.text` *Package*

Example 18.1 illustrates the use of the get methods in the `Locale` class. The call to the `getDefault()` method at (1) returns the default locale. The method call `locNO.getDisplayCountry()` returns the country display name (`Norwegian`) of the Norwegian locale according to the default locale (which in this case is United States), whereas the method call `locNO.getDisplayCountry(locFR)` returns the country display name (`Norvège`) of the Norwegian locale according to the French locale.

Example 18.1 also illustrates that an application can change its default locale pro-grammatically at any time. The call to the `setDefault()` method at (2) sets the default locale to that of Germany. The name of the Norwegian locale is displayed according to this new default locale.

Example 18.1 *Understanding Locales*

Click here to view code image

```
import java.util.Locale;
public class LocalesEverywhere {

  public static void main(String[] args) {

    Locale locDF = Locale.getDefault();                  // (1)
    Locale locNO =  new Locale("no", "NO");              // Locale: Norwegian/Norw
    Locale locFR =  new Locale("fr", "FR");              // Locale: French/France

    System.out.println("Default locale is: " + locDF.getDisplayName());
    System.out.println("Display country (language) for Norwegian locale:");

    System.out.printf("In %s: %s (%s)%n", locDF.getDisplayCountry(),
        locNO.getDisplayCountry(locDF), locNO.getDisplayLanguage(locDF));

    System.out.printf("In %s: %s (%s)%n", locNO.getDisplayCountry(),
        locNO.getDisplayCountry(locNO), locNO.getDisplayLanguage(locNO));

    System.out.printf("In %s: %s (%s)%n", locFR.getDisplayCountry(),
        locNO.getDisplayCountry(locFR), locNO.getDisplayLanguage(locFR));

    System.out.println("\nChanging the default locale.");
    Locale.setDefault(Locale.GERMANY);               // (2) Locale: German/Ger
    locDF = Locale.getDefault();
    System.out.println("Default locale is: " + locDF.getDisplayName());
    System.out.printf("Interpreting %s locale information in %s locale.%n",
                      locNO.getDisplayName(), locDF.getDisplayName());
  }
}
```

Output from the program:

Click here to view code image

```
Default locale is: English (United States)
Display country (language) for Norwegian locale:
In United States: Norway (Norwegian)
In Norway: Norge (norsk)
In France: Norvège (norvégien)

Changing the default locale.
Default locale is: Deutsch (Deutschland)
Interpreting Norwegisch (Norwegen) locale information in Deutsch (Deutschland)
locale.
```

18.2 Properties Files

Applications need to customize their behavior and access information about the runtime environment. This could be about getting configuration data to run the application, tailoring the user interface to a specific locale, accessing the particulars for a database connection, customizing colors and fonts, and many other situations. It is not a good idea to hard-code such information in the application, but instead making it available externally so that the application can access it when needed, and where it can be modified without having to compile the application code.

Creating a Property List in a Properties File

A *property list* contains *key–value pairs* that designate *properties*, where a key is associated with a value, analogous to the entries in a map. Such a list can be created in a *properties file*, where a key–value pair is defined on each line according to the following syntax:

```
<key> = <value>
```

Following are some examples of properties that define information about the Java SE 17 Developer exam:

Click here to view code image

```
cert.name = OCP, Java SE 17 Developer
exam.title = Java SE 17 Developer
exam.number = 1Z0-829
```

Alternatively, we can use the colon (`:`) instead of the equals sign (`=`). In fact, whitespace can also be used to separate the key and the value. Any leading whitespace on a line is ignored, as is any whitespace around the equals sign (`=`).

Both the key and the value in a properties file are interpreted as `String` objects. The value string comprises all characters on the right-hand side of `=`, beginning with the first non-whitespace character and ending with the last non-whitespace character. Here are some examples:

Click here to view code image

```
company = GLOBUS
greeting=Hi!
        gratitude               =Thank you!
```

We do not need to escape the metacharacter `=` or `:` in a value string. However, key names can be specified by escaping any whitespace in the name with the backslash character (`\`). Metacharacters `=` and `:` in a key name can also be escaped in the same way. In the first example below, the key string is `"fake smiley"` and the value string is `":=)"`. In the second example below, the key string is `":=)"` and the value string is `"fake smiley"`.

```
fake\ smiley = :=)
\:\=) = fake smiley
```

A backslash (`\`) at the end of a line can also be used to break the property specification across multiple lines, and any leading whitespace on a continuation line is ignored. The value in the following property specification is `"Au revoir!"` and not `"Au revoir!"`:

Click here to view code image

```
"Au            revoir!":

farewell = Au \
           revoir!
```

If only the key is specified and no value is provided, the empty string (`""`) is returned as the value, as in the following example:

```
KeyWithNoValue
```

If necessary, any conversion of the value string must be explicitly done by the application.

In the case of a duplicate key in a properties file, the last key–value pair with a duplicate key supersedes any previous resource specifications with this key.

If the first non-whitespace character on a line is the hash sign (`#`), the whole line is treated as a comment, and thereby ignored—as are blank lines. The exclamation mark (`!`) can also be used for this purpose.

```
# Any comments so far?
```

If the character encoding of a properties file does not support Unicode characters, then these must be specified using the `\uxxxx` encoding in the file. For example, to specify the property `euro = €`, we can encode it as:

```
euro = \u20AC
```

Since a properties file is a text file, it can conveniently be created in a text editor, and appropriate file permissions set to avoid unauthorized access. Although the name of a properties file can be any valid file name, it is customary to append the file extension `.properties`.

The Java Standard library provides classes to utilize the properties defined in a properties file.

- The `java.util.ResourceBundle` class can be used to implement a *resource bundle* based on the properties in a property (resource) file (p. 1102).
- The `java.util.Properties` class can be used to implement a `Properties` *table* based on the properties defined in a properties file—but this will not be discussed here, as this class is beyond the scope of this book.

18.3 Bundling Resources

Locale-specific data (messages, labels, colors, images, etc.) must be customized according to the conventions and customs for each locale that an application wishes to support. A *resource bundle* is a convenient and efficient way to store and retrieve locale-specific data in an application. The abstract class `java.util.ResourceBundle` is the key to managing locale-specific data in a Java application.

Resource Bundle Families

A *resource bundle* essentially defines *key–value pairs* that are associated with a specific locale. The synonyms *resources* and *properties* are also used for *key–value pairs*, depending on how a resource bundle is implemented. The application can retrieve locale-specific resources from the appropriate resource bundle.

A specific naming convention is used to create resource bundles. This naming convention allows resource bundles to be associated with specific locales, thereby defining *resource bundle families*. All bundles in a resource bundle family have a common *base name*, along with other locale-specific extensions, but the language and country extensions are the important ones to consider. Best practice is to use the following name for a resource bundle that defines all country-specific resources:

Click here to view code image

```
baseName_languageCode_countryCode
```

and to use the following name for a resource bundle that defines all language-specific resources:

```
baseName_languageCode
```

In addition, it is also recommended to include a *default resource bundle* that has only the common base name. The underscore (_) is mandatory. The extensions with the language and country codes are according to locale conventions discussed earlier (p. 1096). The following resource bundle family, used in Example 18.3, has the common base name `BasicResources`:

Click here to view code image

`BasicResources`	Default resource bundle
`BasicResources_no_NO`	Resource bundle for Norwegian (Norway) locale
`BasicResources_fr`	Resource bundle for all locales with French language

An application need not store all resources associated with a locale in a single resource bundle family. The resources can be organized in different resource bundle families for a given locale.

This naming scheme allows a resource bundle to be associated with a specific locale. Note that an application provides a version of each resource bundle for every locale supported by the application, unless they are shared. An example of such sharing is shown above: the locales for France and French-Canada both share the `BasicResources_fr` resource bundle. We have not defined separate `BasicResources` bundles for these two locales. We will have more to say on this matter later when we discuss how resources are located for a given locale in a resource bundle family.

Creating Resource Bundles

A resource bundle can be a *property resource file* (as shown in Example 18.2) or a *resource bundle class*, which is a *subclass of the abstract class* `java.util.ResourceBundle`. A resource bundle class can be used to implement resources that are not strings. We will not discuss resource bundle classes here as they are beyond the scope of this book.

Property Resource Files

A *property resource file* is a *properties file*, in which each line defines a *key–value pair* that designates a *property*. The discussion on properties files (p. 1100) also applies to a property resource file. However, note that a property resource file must be named according to the resource bundle naming scheme outlined above, and has the mandatory file extension `.properties` in addition.

Example 18.2 shows the three property resource files used by Example 18.3. The property resource file `BasicResources.properties` defines the default resource bundle for the resource bundle family `BasicResources` that applies for all locales. The property resource files `BasicResources_no_NO.properties` and `BasicResources_fr.properties` define resource bundles for the Norwegian (Norway) locale and the French locale, respectively. Note the appending of the language/country code to the bundle family name using the underscore (_).

The first line in each of the three property resource files in Example 18.2 is a comment documenting the name of the file. Typically, the property resource files are placed in a directory, usually called `resources`, which is in the same location as the application.

Note also that we use the same key name for a property in *all* property resource files of a resource bundle family. The greeting, gratitude, and the farewell messages are designated by their respective key names in all property resource files. In Example 18.2, the astute reader will notice that the key name `company` is only specified in the default property resource file `BasicResources.properties`, but not in the other property resource files. If the default resource bundle for a resource bundle family is provided, it is always in the search path when locating the value associated with a key.

Modifying an existing property resource file or adding a new property resource file may not require recompiling of the application, depending on the implication of the changes made to the property resource file.

Example 18.2 *The* `BasicResources` *Bundle Family (See Also Example 18.3)*

Click here to view code image

```
# File: BasicResources.properties
company = GLOBUS
greeting = Hi!
gratitude = Thank you!
farewell = See you!
```

Click here to view code image

```
# File: BasicResources_no_NO.properties
greeting = Hei!
gratitude = Takk!
farewell = Ha det!
```

Click here to view code image

```
# File: BasicResources_fr.properties
greeting = Bonjour!
gratitude = Merci!
farewell = Au revoir!
```

Locating, Loading, and Searching Resource Bundles

Once the necessary resource bundle families have been specified, the application can access the resources in a specific resource bundle family for a particular locale using the services of the `java.util.ResourceBundle` class. First, a locale-specific resource bundle is *created* from a resource bundle family using the `getBundle()` method—an elaborate process explained later in this section.

> Click here to view code image
>
> ```
> static ResourceBundle getBundle(String baseName)
> static ResourceBundle getBundle(String baseName, Locale locale)
> ```
>
> Return a resource bundle using the specified base name for a resource bundle family, either for the default locale or for the specified locale, respectively.
>
> The resource bundle returned by this method is *chained* to other resource bundles (called *parent resource bundles*) that are searched if the key-based lookup in this resource bundle fails to find the value of a resource.
>
> An unchecked `java.util.MissingResourceException` is thrown if no resource bundle for the specified base name can be found.
>
> Also, it should be noted that bundles are loaded by the classloader, and thus their bundle names are treated exactly like fully qualified class names; that is, the package name needs to be specified when a resource bundle is placed in a package, such as `"resources.BasicResources"`.
>
> If the resource bundle found by the `getBundle()` method was defined as a *property resource file*, its contents are read into an instance of the concrete class `Prop-ertyResourceBundle` (a subclass of the abstract `ResourceBundle` class) and this instance is returned.
>
> The resource bundle instances returned by the `getBundle()` methods are immutable and are cached for reuse.
>
> Click here to view code image
>
> ```
> Object getObject(String key)
> String getString(String key)
> ```
>
> The first method returns an object for the given key from this resource bundle.
>
> The second method returns a string for the given key from this resource bundle. This is a convenience method, if the value is a string. Calling this method is equivalent to `(String) getObject(key)`. A `ClassCastException` is thrown if the object

> found for the given key is not a string.
>
> An unchecked `java.util.MissingResourceException` is thrown if no value for the key can be found in this resource bundle or any of its parent resource bundles. A `NullPointerException` is thrown if the key is `null`.
>
> ```
> Set<String> keySet()
> ```
>
> Returns a `Set` of all keys contained in this `ResourceBundle`.
>
> ```
> Locale getLocale()
> ```
>
> Returns the locale of this resource bundle. The locale is derived from the naming scheme for resource bundles.

We will use Example 18.3 to illustrate salient features of localizing an application using resource bundles. The application has the following data, which should be localized:

```
Company: GLOBUS
Greeting: Hi!
Gratitude: Thank you!
Farewell: See you!
```

The example illustrates how the application can localize this data for the following locales: default (`"en_US"`), Norway (`"no_NO"`), French-Canada (`"fr_CA"`), and France (`"fr_FR"`). The company name is the same in every locale. The greeting, gratitude, and farewell messages are all grouped in one resource bundle family (`BasicResources`). For each locale, the application reads this data from the appropriate resource bundles and prints it to the terminal (see the output from Example 18.3).

Example 18.3 *Using Resource Bundles (See Also Example 18.2)*

Click here to view code image

```
import java.util.Locale;
import java.util.ResourceBundle;
public class UsingResourceBundles {

  // Supported locales:                                                   // (
  public static final Locale[] locales = {
    Locale.getDefault(),                                // Default: US (Englis
    new Locale("no", "NO"),                             // Norway (Norwegian)
    Locale.FRANCE,                                      // France (French)
    Locale.CANADA_FRENCH                                // Canada (French)
  };

  // Localized data from property resource files:                         // (
  private static String company;
```

```
  private static String greeting;
  private static String gratitude;
  private static String farewell;

  public static void main(String[] args) {                                // (
    for (Locale locale : locales) {
      setLocaleSpecificData(locale);
      printLocaleSpecificData(locale);
    }
  }

  private static void setLocaleSpecificData(Locale locale) {              // (
    // Get resources from property resource files:
    ResourceBundle properties =
        ResourceBundle.getBundle("resources.BasicResources", locale);     // (
    company = properties.getString("company");                            // (
    greeting = properties.getString("greeting");
    gratitude = properties.getString("gratitude");
    farewell = properties.getString("farewell");
  }

  private static void printLocaleSpecificData(Locale locale) {            // (
    System.out.println("Resources for " + locale.getDisplayName() + " locale:"
    System.out.println("Company: " + company);
    System.out.println("Greeting: " + greeting);
    System.out.println("Gratitude: " + gratitude);
    System.out.println("Farewell: " + farewell);
    System.out.println();
  }
}
```

Output from running the program:

Click here to view code image

```
Resources for English (United States) locale:
Company: GLOBUS
Greeting: Hi!
Gratitude: Thank you!
Farewell: See you!

Resources for Norwegian (Norway) locale:
Company: GLOBUS
Greeting: Hei!
Gratitude: Takk!
Farewell: Ha det!

Resources for French (France) locale:
Company: GLOBUS
Greeting: Bonjour!
Gratitude: Merci!
Farewell: Au revoir!

Resources for French (Canada) locale:
Company: GLOBUS
Greeting: Bonjour!
Gratitude: Merci!
Farewell: Au revoir!
```

In Example 18.3, the static method `getBundle()` of the `ResourceBundle` class is called at (5) to create a resource bundle for the specified locale from the resource bundle family with the base name `BasicResources`. In Example 18.3, the resource bundle family is located in the `resources` directory, which is also the location of the package with the same name.

Click here to view code image

```
String company;
...
ResourceBundle properties =
    ResourceBundle.getBundle("resources.BasicResources", locale);    // (5)
company = properties.getString("company");                           // (6)
```

Since all the resource bundles in the resource bundle family `BasicResources` are property resource files, both the key and the value are `String` objects. It is convenient to use the `getString()` method in the `ResourceBundle` class to do lookup for resources, as shown at (6). The key is passed as an argument. Searching the resource for a given key is explained later in this section.

Locating Locale-Specific Resources

A broad outline of the steps is given below to locate and create the resource bundle (and its parent bundles) returned by the `getBundle()` static method of the abstract class `java.util.ResourceBundle` for a specific locale. For the nitty-gritty details of locating resource bundles, we recommend consulting the Java SE API documentation for the `java.util.ResourceBundle` class.

Exactly which resource bundle (and its parent bundles) will be returned by the `getBundle()` static method depends on the following factors:

- The *resource bundles included in the resource bundle family* specified by its fully qualified base name in the call to the `getBundle()` method
- The *specified locale* in the call to the `getBundle()` method
- The *current default locale* of the application

For the explanation given below, assume that the default local is `"en_US"` and the following resource files (from Example 18.2) in the resource bundle family with the base name `BasicResources` reside in a directory named `resources`:

Click here to view code image

```
BasicResources.properties
BasicResources_no_NO.properties
BasicResources_fr.properties
```

Also assume the following call is made to the `getBundle()` method to retrieve the resource bundle (and its parent resource bundles) for the `baseName` bundle family and the specified locale:

Click here to view code image

```
ResourceBundle resources = ResourceBundle.getBundle(baseName, specifiedLocale)
```

Step 1: Create a list of candidate bundle names based on the specified locale.

The `getBundle()` method first generates a list of *candidate bundle names* by appending the attributes of the locale argument (specified language code, specified country code) to the base name of the resource bundle family that is passed as an argument. Typically, this list would have the following candidate bundle names, where the order in which the names are generated is important in locating resources:

Click here to view code image

```
baseName_specifiedLanguageCode_specifiedCountryCode
baseName_specifiedLanguageCode
```

For example, if the base name is `BasicResources` and the locale that is passed as an argument is `"fr_CA"` in the call to the `getBundle()` method, the list of candidate bundle names generated would be as follows:

```
BasicResources_fr_CA
BasicResources_fr
```

Note that more specific bundles are higher in this list, and more general ones are lower in the list. The bundle `BasicResources_fr_CA` is more specific than the bundle `BasicResources_fr`, as the former is only for France (French), whereas the latter is for all French-speaking countries.

A candidate bundle name in this list is a *parent* resource bundle for the one above it.

Step 2: Find the result bundle that can be instantiated in the candidate bundle list.

The `getBundle()` method then iterates over the list of candidate bundle names from the beginning of the list to find the *first* one (called the *result bundle*) for which it can instantiate an actual resource bundle.

Finding the result bundle depends on whether it is possible to instantiate a resource bundle class or load the contents of a property resource file into an instance of the `PropertyResourceBundle` class, where the resource bundle class or the property resource file has the same name as the candidate bundle name.

In our example regarding the `"fr_CA"` locale, the candidate resource name `BasicResources_fr_CA` in the list of candidate resource names does not qualify as a result resource bundle because it cannot be created based on any resource bundles in the resource bundle family `BasicResources`. Only the candidate resource name `BasicResources_fr` can be created and is the result resource bundle, as there is a resource bundle named `BasicResources_fr.properties` in the resource bundle family.

Once a result resource bundle has been found, its *parent chain* is constructed and returned (p. 1109). In our example with the `"fr_CA"` locale, the parent chain for the result source bundle named `BasicResources_fr.properties` is created and returned.

Step 3: If no result resource bundle is found in Step 2, the search for a result bundle is conducted with the default locale.

It is possible that no result resource bundle is found in the previous step. Only in that case is the search for a result resource bundle repeated with the current *default locale* (default language code, default country code). A new list of candidate bundle names is generated, which will typically include the following names, and is instantiated to find a result bundle name:

Click here to view code image

```
baseName_defaultLanguageCode_defaultCountryCode
baseName_defaultLanguageCode
```

In our example, this second step is not performed since a result resource bundle (`BasicResources_fr`) was found in the previous step.

Once a result resource bundle using the default locale has been found, its *parent chain* is constructed and returned (p. 1109).

Step 4: If no result resource bundle is found in Step 3 using the current default locale either, an attempt is made to instantiate the default resource bundle designated by `baseName`*.*

If successful, this instantiation of the default resource bundle is returned by the `getBundle(baseName, specifiedLocale)` method.

In our example, this step with the bundle name `BasicResources` is not performed since a result resource bundle (`BasicResources_fr`) was already found.

Step 5: If the steps above do not yield a result resource bundle, an unchecked `java.util.MissingResourceException` *is thrown.*

In our example, no exception is thrown since a result resource bundle (`BasicResources_fr`) was found in Step 2.

Constructing the Parent Chain of a Result Resource Bundle

Once a result resource bundle is identified (see Step 2 and Step 3 in the previous section), a *parent chain* for the result source bundle is constructed and returned by the `getBundle()` method as follows:

- All candidate bundle names that are below the result bundle name in the candidate bundle list are iterated over, and those candidate bundles that can be instantiated are *chained* in the order in which they appear in the list.

  In our example, the result resource bundle `BasicResources_fr` was found last in the list of candidate bundle names, and therefore does not get any parent according to this step.

- Lastly, an attempt is also made to chain the *default resource bundle* (with the base name) into the parent chain of the result resource bundle which is then returned by the `getBundle(baseName, specifiedLocale)` method.

In our example, the default resource bundle `BasicResources` is provided and is therefore set as the parent bundle to the result resource bundle `Basic-Resources_fr`.

Click here to view code image

```
BasicResources_fr.properties
BasicResources.properties (parent)
```

This will result in the parent chain being searched, if necessary, when a lookup is performed for a resource in this result bundle for the `"fr_CA"` locale, where the most specific bundle (`BasicResources_fr.properties`) is researched first and the most general one (`BasicResources.properties`) last.

Note that an attempt is also made to include the default resource bundle when no result resource bundle can be found in the candidate bundle list, as shown in Step 4 earlier.

Best practice recommends including a default resource bundle as *fallback* in a resource bundle family. This bundle will then always be searched last to find the value associated with a key.

Table 18.5 shows the result resource bundles with the parent bundle chain from the resource bundle family `BasicResources` that will be located and loaded for each locale supported by Example 18.3. We have shown property resource file names in the table and indicated the parent resource bundle. Actually, resource bundle classes are instantiated at runtime, based on the contents of the property resource files. Note that the resource bundle `BasicResources_fr.properties` is searched for both the `"fr_FR"` locale and the `"fr_CA"` locale.

Table 18.5 *Result and Parent Bundles (See Example 18.3)*

Specified locale	**Step that creates the result bundle**	**Result bundles with the parent chain:**
`Locale("en","US")`	*Step 4*	`BasicResources.properties`
`Locale("no","NO")`	*Step 2*	`BasicResources_no_NO.properties` `BasicResources.properties` *(parent)*
`Locale("fr","FR")`	*Step 2*	`BasicResources_fr.properties` `BasicResources.properties` *(parent)*
`Locale("fr","CA")`	*Step 2*	`BasicResources_fr.properties` `BasicResources.properties` *(parent)*

Searching in Resource Bundles

Example 18.4 is instructive in understanding which resource bundles for a given locale will be searched when performing key-based lookups for locale-specific values. Each resource bundle in the resource family `MyResources` has only one key, and the name of the key is different in all resource bundles. The value of the key in each file is the name of the resource bundle. The program prints the value of all the keys found in the resource bundles for a given locale. The method `keySet()`, called at (1), returns a set with all the keys from the resource bundles that can be searched for the locale specified in the `getBundle()` method. Any value written out will be the name of the resource bundle that is searched. Since a key set is returned, the key order in the set is not guaranteed. However, this key set is converted to a sorted key set to reflect the hierarchy of resource bundles for each locale.

From the program output, we can see that the default resource bundle is always searched (for all the locales in Example 18.4). Only if no corresponding resource bundle for the given locale can be found in the resource bundle family will the resource bundle for the current default locale be searched (`"no_NO"` locale in Example 18.4). For a language-country locale, both the country and the language resource bundles are included, if any of them are in the resource bundle family (the `"fr_CA"` locale in Example 18.4). For a language locale, only the language resource bundle is included, if it is in the resource bundle family (`"fr"` locale in Example 18.4).

Example 18.4 *Locating Resource Bundles*

Click here to view code image

```
# MyResources_fr_CA.properties
file1 = MyResources_fr_CA
```

Click here to view code image

```
# MyResources_fr.properties
file2 = MyResources_fr
```

Click here to view code image

```
# MyResources_en_US.properties
file3 = MyResources_en_US
```

Click here to view code image

```
# MyResources_en.properties
file4 = MyResources_en
```

Click here to view code image

```
# MyResources.properties
file5 = MyResources
```

Click here to view code image

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.TreeSet;
```

```
public class LocatingBundles {
  public static void main(String[] args) {

    Locale[] locales = {
        new Locale("no", "NO"),                             // Norway
        Locale.CANADA_FRENCH,                               // Canada (French)
        Locale.FRENCH,                                      // French
        Locale.getDefault(),                                // Default: en_US
    };

    for (Locale locale: locales) {
      System.out.println("Locating resource bundles for " + locale + " locale:
      ResourceBundle resources = ResourceBundle.getBundle("resources.MyResourc
                                                          locale);
      for (String key : new TreeSet<>(resources.keySet())) {               //
        System.out.println(resources.getString(key));
      }
      System.out.println();
    }
  }
}
```

Output from the program:

Click here to view code image

```
Locating resource bundles for no_NO locale:
MyResources_en_US
MyResources_en
MyResources

Locating resource bundles for fr_CA locale:
MyResources_fr_CA
MyResources_fr
MyResources

Locating resource bundles for fr locale:
MyResources_fr
MyResources

Locating resource bundles for en_US locale:
MyResources_en_US
MyResources_en
MyResources
```

Review Questions

18.1 Given the following resource bundle in the `pkg` package directory:

Click here to view code image

```
# File: MyResources_en_US.properties
greeting = Howdy!
gratitude = Thank you!
farewell = See ya!
farewell = Bye!
```

Assume that the current default locale is `"en_US"`. What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.*;
public class TestResourceBundles {
  public static void main(String[] args) {
    ResourceBundle resources = ResourceBundle.getBundle("pkg.MyResources
                                                        Locale.FRANCE);
    for (String key : resources.keySet()) {
      System.out.println(resources.getString(key));
    }
  }
}
```

Select the one correct answer.

(a) The program output will contain the following lines:

```
Howdy!
Bye!
Thank you!
```

a. The program output will contain the following lines:

```
Howdy!

 See ya!
 Thank you!
```

b. When run, the program will throw a `MissingResourceException`.

c. When run, the program will terminate normally without printing anything.

18.2 Given the following three resource bundles in the `pkg3` package:

Click here to view code image

```
# MyResources_en_US.properties
farewell = See ya!
```

```
# MyResources_en.properties
farewell = Have a good one!
```

```
# MyResources.properties
farewell = Goodbye!
```

Assume that the current default locale is `"en_US"`. What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.*;
public class TestResourceBundles3 {
  public static void main(String[] args) {
    ResourceBundle resources = ResourceBundle.getBundle("pkg3.MyResource
                                                        Locale.ENGLISH);
    System.out.println(resources.getString("farewell"));
  }
}
```

Select the one correct answer.

a. `See ya!`
b. `Have a good one!`
c. `Goodbye!`
d. When run, the program will throw a `MissingResourceException`.
e. When run, the program will terminate normally without printing anything.

18.3 Given the following code:

Click here to view code image

```
import java.util.*;
public class RQ4_2 {
  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("resources", new Locale
    for (String key : new TreeSet<String>(rb.keySet())) {
      System.out.print(key+"="+rb.getString(key)+" ");
    }
  }
}
```

and the file `resources.properties`

```
k1=a
k2=b
```

and the file `resources_en.properties`

```
k1=c
```

and the file `resources_en_GB.properties`

```
k1=d
k2=e
```

What is the result?
Select the one correct answer.

a. `k1=d k2=e`
b. `k1=c k2=b`
c. `k1=a k2=b`
d. `k1=d k2=e`
e. The program will throw an exception at runtime.

18.4 Given the following code:

Click here to view code image

```
import java.util.*;
public class RQ4_3 {
  public static void main(String[] args) {
    Locale.setDefault(new Locale("ru"));
    ResourceBundle rb = ResourceBundle.getBundle("resources",new Locale(
    for (String key : new TreeSet<String>(rb.keySet())) {
      System.out.print(key+"="+rb.getString(key)+" ");
      }
    }
  }
```

and the file `resources.properties`

```
k1=a
k2=b
```

and the file `resources_en.properties`

```
k1=c
```

and the file `resources_ru.properties`

```
k1=A
k2=B
```

What is the result?
Select the one correct answer.

a. `k1=c k2=B`
b. `k1=c k2=b`
c. `k1=A k2=B`
d. The program will throw an exception at runtime.

18.4 Core API for Formatting and Parsing of Values

The abstract class `Format` and its subclasses in the `java.text` package provide support for formatting and parsing of dates, times, numbers, currencies, and percentages (Figure 18.1).

A *formatter* has two primary functions. The first is to create a human-readable text representation of a value, called *formatting*. The second is to create a value from a character sequence, such as a string, containing a text representation of the value, called *parsing*.

In the rest of this chapter, we emphasize the *locale-sensitive* nature of formatting and parsing of different kinds of values.

- *Formatting numbers, currencies, and percentages (p. 1116)*

  The support for formatting numbers, currencies, and percentages is provided by the abstract class `java.text.NumberFormat` and its subclasses `java.text.DecimalFormat` and `java.text.CompactNumberFormat`.

- *Formatting dates and times (p. 1127)*

The support provided for formatting dates and times by the abstract class `java.text.DataFormat` and its subclass `java.text.SimpleFormat` is aimed at objects of the `java.util.Date` legacy class, and is *not* covered in this book.

Support for the new Date and Time API is provided by the `java.time` package, which is covered extensively in Chapter 17, p. 1023. The class `java.time.format.DateTimeFormatter` provides the support for formatting date and time values created using the new Date and Time API.

- *Formatting messages (p. 1139)*

  The support for formatting *messages* containing numbers, currencies, dates, and times is provided by the concrete class `java.text.MessageFormat`. The class `java.text.ChoiceFormat` allows a choice of formatting dependent on numerical values, and is typically used for *conditional formatting* of messages.

  The support provided by the `MessageFormat` class for date and time values is aimed at objects of the legacy class `Date`, but there are a number of ways that the `MessageFormat` class can be utilized with formatting objects of the new Date and Time API, and they are discussed later (p. 1139).

18.5 Formatting and Parsing Number, Currency, and Percentage Values

The abstract class `java.text.NumberFormat` and its subclasses `java.text.Decimal-Format` and `java.text.CompactNumberFormat` provide methods for locale-sensitive formatting and parsing of *number*, *currency*, and *percentage* values.

Static Factory Methods to Create a Formatter

The abstract class `NumberFormat` provides static factory methods for creating locale-sensitive formatters for number, currency, and percentage values. However, the locale cannot be changed after the formatter is created. The factory methods return instances of the concrete classes `DecimalFormat` and `CompactNumberFormat` for formatting number, currency, and percentage values.

> Click here to view code image
>
> ```
> static NumberFormat getNumberInstance()
> static NumberFormat getNumberInstance(Locale locale)
> ```
>
> Return a general formatter for numbers—that is, a number formatter.
>
> Click here to view code image
>
> ```
> static NumberFormat getCurrencyInstance()
> static NumberFormat getCurrencyInstance(Locale locale)
> ```
>
> Return a formatter for currency values—that is, a currency formatter.
>
> Click here to view code image
>
> ```
> static NumberFormat getPercentInstance()
> static NumberFormat getPercentInstance(Locale locale)
> ```
>
> Return a formatter for percentages—that is, a percentage formatter.
>
> Click here to view code image
>
> ```
> static NumberFormat getCompactNumberInstance()
> static NumberFormat getCompactNumberInstance(Locale locale,
>                                              NumberFormat.Style formatStyle)
> ```
>
> Return a compact number formatter with the default compact format style or a specific compact format style, respectively (p. 1120). For example, the value `2_345_678` can be formatted by a compact number formatter as `"2M"` with the `SHORT` compact form style and as `"2 million"` with the `LONG` compact form style, respectively, for the `US` locale.
>
> In all cases, a locale can be specified to localize the formatter.

Formatting Number, Currency, and Percentage Values

A number formatter can be used to format a `double`, a `long` value, or an object. The abstract class `NumberFormat` provides the following concrete methods for this purpose. Depending on the number formatter, the formatting is locale-sensitive, determined by the default or a specific locale.

> ```
> String format(double d)
> String format(long l)
> ```

> Formats the specified number and returns the resulting string.
>
> Click here to view code image
>
> ```
> String format(Object obj)             // inherited from the Format class.
> ```
>
> Formats the specified object and returns the resulting string. For example, it can be used to format `BigInteger` and `BigDecimal` numbers.

The following code shows how we can create a *number formatter* for the Norwegian locale and one for the US locale, and use them to format numbers according to rules of the locale. The number formatted is a `double` (1a) or a `BigDecimal` (1b), giving the same results. Note that the grouping of the digits and the decimal separator used in formatting is according to the locale.

Click here to view code image

```
double num = 12345.6789;                                    // (1a)
// BigDecimal num = new BigDecimal("12345.6789");           // (1b)

Locale locNOR = new Locale("no", "NO");                     // Norway
NumberFormat nfNOR = NumberFormat.getNumberInstance(locNOR);
System.out.println(nfNOR.format(num));                      // 12 345,679

NumberFormat nfUS = NumberFormat.getNumberInstance(Locale.US);
System.out.println(nfUS.format(num));                       // 12,345.679
```

The following code shows how we can create a *currency* formatter for the Norwegian locale, and use it to format currency values according to this locale. Note the currency symbol and the grouping of the digits, with the amount being rounded to two decimal places. Also note that the delimiter between the currency symbol and the first digit is a *non-breaking space* (*nbsp*), having the unicode `\u00a0`; that is, it is not a normal space (`\u0020`). The grouping of digits is also with a nbsp. Such use of a nbsp is also locale-specific—formatting for the US locale has no nbsp, as can be seen below.

Click here to view code image

```
NumberFormat cfNOR = NumberFormat.getCurrencyInstance(locNOR);
String formattedCurrStr = cfNOR.format(num);
System.out.println(formattedCurrStr);                // kr 12 345,68 (with 2 nbs

NumberFormat cfUS = NumberFormat.getCurrencyInstance(Locale.US);
String formattedCurrStrUS = cfUS.format(num);
System.out.println(formattedCurrStrUS);              // $12,345.68
```

The value 1.0 is equivalent to 100%. A value is converted to a percentage by multiplying it by 100, and by default rounding it to an integer. By default, the result is rounded up only if the decimal part is *equal to or greater than* 0.5 after multiplying by 100. However, both the number of decimal places and the rounding behavior for the percentage value can be changed (p. 1122).

The following code shows how we can create a *percentage* formatter for the Norwegian locale, and use it to format percentage values according to this locale. Also note that the delimiter between the percentage number and the currency symbol is a *non-breaking space*. Again, such use of a nbsp is also locale-specific—formatting percentages for the US locale has no nbsp, as can be seen below. Note the rounding of the percentage values.

Click here to view code image

```
double rebate = 0.746;
NumberFormat pfNOR = NumberFormat.getPercentInstance(locNOR);
String formattedPStr = pfNOR.format(rebate);
System.out.println(formattedPStr);                  // 75 %  (with nbsp)

NumberFormat pfUS = NumberFormat.getPercentInstance(Locale.US);
String formattedPStrUS = pfUS.format(rebate);
System.out.println(formattedPStrUS);                // 75%

System.out.println(pfUS.format(0.745));             // 74%
```

Accounting Currency Formatting

By default, the minus sign (-) is used as a prefix when formatting negative numbers using a currency formatter returned by the `NumberFormat.getCurrencyInstance()` method.

Click here to view code image

```
NumberFormat df0 = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(df0.format(-9.99));             // -$9.99
```

However, in accountancy, it is a common practice to enclose a negative currency value in parentheses, `()`. For example, `($9.99)` is interpreted as `-$9.99`. One way to format negative currency values with parentheses for a given locale is to create the locale so that it allows this style for currency formatting. The code below illustrates how this can be achieved for the US locale.

A US locale is created at (1) below that formats negative currency values with parentheses. The `Locale.forLanguageTag()` method creates a locale from the specified language tag `en-US-u-cf-account` that consists of several subtags separated by a hyphen:

- `en`: Represents the English language

- `US`: Represents the United States
- `u`: Indicates that a *Unicode locale/language extension* is specified as a *key–value* pair
- `cf-account`: The key–value pair, where the key `cf` stands for *currency format*, and the value `account` means to use parentheses for negative numbers

The language tag `en-US-u-cf-standard` will result in a US locale that is the same as `Locale.US`. For further details, kindly consult the Unicode Locale Data Markup Language (LDML).

The currency formatter created at (2) for the locale created at (1) will now use parentheses for negative currency values.

Click here to view code image

```
Locale loc = Locale.forLanguageTag("en-US-u-cf-account");    // (1)
NumberFormat df = NumberFormat.getCurrencyInstance(loc);     // (2)
System.out.println(df.format(-9.99));                        // ($9.99)
```

Parsing Strings to Number, Currency, and Percentage Values

A *number* formatter can be used to parse strings that contain text representations of *numeric* values. The abstract class `NumberFormat` provides the following concrete methods for this purpose.

> Click here to view code image
>
> ```
> Number parse(String str) throws ParseException
> ```
>
> Parses the text in the specified string from the beginning of the string to produce a `Number`. No leading whitespace is allowed. Any trailing characters after the parsed text are ignored. This method throws the checked `java.text.Parse-Exception` if unsuccessful.
>
> Click here to view code image
>
> ```
> void setParseIntegerOnly(boolean intOnly)
> boolean isParseIntegerOnly()
> ```
>
> Set or get the status if this formatter should only parse integers.

The class `DecimalFormat` specifically provides the following concrete methods to customize the formatter to parse `BigDecimal` numbers.

> Click here to view code image
>
> ```
> void setParseBigDecimal(boolean newValue)
> boolean isParseBigDecimal()
> ```
>
> Set or get the status if this formatter should parse `BigDecimal` numbers.

The code calling the `parse()` method must be prepared to handle a checked `Parse-Exception`—for brevity, exception handling is omitted in the code below.

The following code shows the Norwegian number formatter from earlier being used to parse strings. At (1), the result is a `long` value because the dot (`.`) in the input string is a delimiter and not a legal character according to the number format used in the Norwegian locale. At (2), the result is a `double` value because the comma (`,`) in the input string is the decimal separator in the Norwegian locale. Note that the print statement prints the resulting number according to the *default* locale (U.S., in this case). For the US locale, the dot (.) and the comma (,) are interpreted as the decimal separator and the group separator, respectively.

Click here to view code image

```
System.out.println(nfNOR.parse("9876.598"));      // (1) 9876
System.out.println(nfNOR.parse("9876,598"));      // (2) 9876.598

System.out.println(nfUS.parse("9876.598"));       // (3) 9876.598
System.out.println(nfUs.parse("9876,598"));       // (4) 9876598
```

In order to parse a string to a `BigDecimal`, we can proceed via a `DecimalFormat` that is customized to parse a `BigDecimal`. The overloaded `parse()` method below returns a `Number`, which is actually a `BigDecimal` in this case.

Click here to view code image

```
DecimalFormat dfUS = (DecimalFormat) nfUS;
dfUS.setParseBigDecimal(true);
BigDecimal bd = (BigDecimal) dfUS.parse("9876,598");
```

The following code demonstrates using a *currency* formatter as a parser. Note that the currency symbol is interpreted according to the locale in the currency formatter.

In the Norwegian locale, the decimal symbol is not a period (`.`) when parsing numbers; it is a *delimiter* in the input string, as can be seen at (1) below. For some locales, a *non-breaking space* (`\u00a0`) may be required between the currency symbol and the first digit, as can be seen at (2).

The grouping symbol in the Norwegian locale is not a space, and therefore, it is interpreted as a delimiter, as can be seen at (3). An explicit nbsp can be used for grouping digits, if necessary, in order to parse the input string, as at (4). For the US locale, no such considerations are necessary, as can be seen at (5).

Click here to view code image

```
System.out.println(cfNOR.parse("kr\u00a09876.59"));         // (1) 9876
System.out.println(cfNOR.parse("kr\u00a09876,59"));         // (2) 9876.59
System.out.println(cfNOR.parse("kr\u00a09 876,59"));        // (3) 9
System.out.println(cfNOR.parse("kr\u00a09\u00a0876,59"));   // (4) 9876.59

System.out.println(cfUS.parse("$9876.59"));                 // (5) 9876.59
```

When parsing percentages, a *nbsp* might be required between the percentage value and the percent sign (`%`) in the input string, as can be seen at (1) below for the Norwegian locale. For parsing a percentage value according to the US locale, no such consideration is necessary, as can be seen at (2). However, the format of the percentage value is according to the locale used by the formatter, as can be seen at (1) and (2).

Click here to view code image

```
System.out.println(pfNOR.parse("15,75\u00a0%"));            // (1) 0.1575
System.out.println(pfUS.parse("25.5%"));                    // (2) 0.255
```

Compact Number Formatting

A *compact number formatter* creates a textual representation that represents the *compact form* of a number. This formatter is an instance of the `CompactNumberFormat` that can be created by calling the `getCompactNumberInstance()` factory method of the `NumberFormat` class. For example, the value `1_000_000` can be formatted in a compact form as `"1M"` or `"1 million"`.

The compact form can be specified by the constants of the enum type `Number-Format.Style`, shown in Table 18.6. The compact form has a suffix, depending on the value of the number and the locale. Suffixes for the US locale are shown in Table 18.6. The compact form of numbers below 1000 is without any suffix.

Table 18.6 *Compact Number Styles*

Styles for compact form	**Suffixes for the US locale in compact form**
`NumberFormat.Style.SHORT` (default)	`T` (Trillion), `B` (Billion), `M` (Million), `K` (Thousand) Examples: `2T`, `2.5M`, `1.5K`

Long number format style	`NumberFormat.Style.LONG`	billion, million, thousand Examples: `2 trillion, 2.5 million, 1.5 thousand`

The code below creates two compact number formatters for the US locale that use the `SHORT` and the `LONG` compact number styles, respectively, to format numbers to their compact form.

Click here to view code image

```
NumberFormat shortCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.SHORT)
NumberFormat longCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.LONG);
```

The compact number formatters are used on different numerical values to create their compact form, as shown in Table 18.7.

Table 18.7 *Formatting Numbers to Compact Form*

Number n	**Compact form returned by `shortCompactFormatter.format(n)` method**	**Compact form returned by `longCompactFormatter.format(n)` method**

`9_400_000`
`9_500_000`
`12_500`
`12_510`
`999`
	`"9M"`
	`"10M"`
	`"12K"`
	`"13K"`
	`"999"`
		`"9 million"`
		`"10 million"`
		`"12 thousand"`
		`"13 thousand"`
		`"999"`

(Max fraction digits = 2)
`9_400_000`
`9_500_000`
`12_500`
`12_510`
`999`
	
	`"9.4M"`
	`"9.5M"`
	`"12.5K"`
	`"12.51K"`
	`"999"`
		
		`"9.4 million"`
		`"9.5 million"`
		`"12.5 thousand"`
		`"12.51 thousand"`
		`"999"`

Click here to view code image

```
System.out.println(shortCompactFormat.format(9_400_000));     // 9M
System.out.println(longCompactFormat.format(9_400_000));      // 9 million
```

The second row in Table 18.7 shows the compact form generated after the number of maximum fraction digits is set to 2 (p. 1122):

Click here to view code image

```
shortCompactFormat.setMaximumFractionDigits(2);
longCompactFormat.setMaximumFractionDigits(2);
```

Note the rounding that takes place depending on the value of the number. By default, `RoundingMode.HALF_EVEN` is used (see Table 18.9).

Compact Number Parsing

The `parse()` method of the compact number formatter can be used to parse a string that contains a compact form to a numerical value.

Click here to view code image

```
try {
  System.out.println(shortCompactFormat.parse("9M"));          // 9000000
  System.out.println(longCompactFormat.parse("9 million"));   // 9000000
} catch (ParseException pe) {
  System.out.println(pe);
}
```

The compact number formatters above are used to parse different compact forms to numerical values, as shown in Table 18.8. Note that parsing requires that the compact form ends in an appropriate suffix; otherwise, the suffix is ignored, as we can see in Table 18.8.

Table 18.8 *Parsing Compact Form to Numbers*

Number formatter	Compact form string `s`	Number returned by `.parse(s)` method
`shortCompactFormatter`	`"9M"`	`9000000`
	`"9.5M"`	`9500000`
	`"2K"`	`2000`
	`"1.5K"`	`1500`
	`"9999"`	`9999`
`longCompactFormatter`	`"9 million"`	`9000000`
	`"9.5 million"`	`9500000`
	`"2 thousand"`	`2000`
	`"1.5 thousand"`	`1500`

Specifying the Number of Digits

The following methods of the `NumberFormat` abstract class and its subclass `Decimal-Format` allow formatting of numbers to be further refined by setting the number of digits to be allowed in the integral and the decimal part of a number. This also applies for `BigDecimal` numbers. However, a concrete number formatter can enforce certain limitations on these bounds. In addition, a rounding mode can be set as explained below.

Click here to view code image

```
void setMinimumIntegerDigits(int n)
int  getMinimumIntegerDigits()

void setMaximumIntegerDigits(int n)
int  getMaximumIntegerDigits()

void setMinimumFractionDigits(int n)
int  getMinimumFractionDigits()

void setMaximumFractionDigits(int n)
int  getMaximumFractionDigits()
```

Set or get the minimum or maximum number of digits to be allowed in the integral or decimal part of a number.

Click here to view code image

```
void setRoundingMode(RoundingMode roundingMode)
```

Sets the *rounding mode* used in this number formatter—that is, how the resulting value is rounded by the formatter. The enum type `java.math.RoundingMode` defines constants for such modes (see Table 18.9).

Table 18.9 *Selected Rounding Modes*

Enum type `java.math.Rounding`	**Description**
`CEILING`	Rounds toward *positive infinity*. `1.1 ->  2` (same as `UP`) `-1.8 -> -1` (same as `DOWN`)
`FLOOR`	Rounds toward *negative infinity*. `1.8 ->  1` (same as `DOWN`) `-1.1 -> -2` (same as `UP`)

`UP`	Rounds *away from zero*. Never *decreases* the *magnitude* of the calculated value.
	`1.1 ->  2`
	`-1.1 -> -2`

`DOWN`	Rounds *toward zero*. Never *increases* the *magnitude* of the calculated value.
	`1.8 ->  1`
	`-1.8 -> -1`

`HALF_UP`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds *up*. This is the same as *normal rounding*.
	Click here to view code image
	` 1.5 ->  2 (same as UP, if discarded fraction is >= 0.5)`
	` 1.4 ->  1 (same as DOWN, if discarded fraction is < 0.5)`
	`-1.4 -> -1 (same as DOWN, if discarded fraction is < 0.5)`
	`-1.5 -> -2 (same as UP, if discarded fraction is >= 0.5)`

`HALF_DOWN`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds *down*.
	Click here to view code image
	` 1.6 ->  2 (same as UP, if discarded fraction is > 0.5)`
	` 1.5 ->  1 (same as DOWN, if discarded fraction is <= 0.5)`
	`-1.5 -> -1 (same as DOWN, if discarded fraction is <= 0.5)`
	`-1.6 -> -2 (same as UP, if discarded fraction is > 0.5)`

`HALF_EVEN`	Rounds toward the nearest neighboring value, unless both neighboring values are equidistant, in which case it rounds toward the even neighbor. This rounding policy is used in floating-point arithmetic in Java.
	Click here to view code image
	` 2.5 ->  2 (same as HALF_DOWN, if digit left of discarded fraction is even)`
	` 1.5 ->  2 (same as HALF_UP, if digit left of discarded fraction is odd)`
	`-1.5 -> -2 (same as HALF_UP, if digit left of discarded fraction is odd)`
	`-2.5 -> -2 (same as HALF_DOWN, if digit left of discarded fraction is even)`

Example 18.5 demonstrates the rounding modes defined by the enum type `java.math.RoundingMode`. The examples in Table 18.9 are computed in Example 18.5. A number format is created at (1) for the US locale. The maximum number of digits in the fraction part is set to 0 at (2); that is, the floating-point value will be rounded to an integer before formatting.

The method `roundIt()` at (3) does the rounding and the formatting when passed a formatter, the maximum number of digits (0) required in the fraction part, the rounding mode, and the two values to round and format. The method sets the maximum number of digits required in the fraction part and the rounding mode at (4) and (5), respectively, in the formatter. The formatting of the values is done by calling the `format()` method at (6) and (7). We encourage checking the output against the rules for rounding for each mode shown in Table 18.9.

Example 18.5 *Rounding Modes*

Click here to view code image

```
import java.math.RoundingMode;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class Rounding {

  public static void main(String[] args) {
    System.out.println(" Rounding:   v1          v2");
    NumberFormat nfmtUS = NumberFormat.getNumberInstance(Locale.US);    // (1
    int maxFractionDigits = 0;                                          // (2
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.CEILING,   1.1, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.FLOOR,     1.8, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.UP,        1.1, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.DOWN,      1.8, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   1.5, 1.4);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   -1.4, -1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, 1.6, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, -1.5, -1.6);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, 2.5, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, -1.5, -2.5);
  }

  static void roundIt(NumberFormat nf, int maxFractionDigits, RoundingMode rMo
                      double v1, double v2) {                 // (3)
    nf.setMaximumFractionDigits(maxFractionDigits);           // (4)
    nf.setRoundingMode(rMode);                                // (5)
    System.out.printf("%9s: ", rMode);
    System.out.printf("%5s -> %2s ", v1, nf.format(v1));      // (6)
    System.out.printf("%5s -> %2s%n", v2, nf.format(v2));     // (7)
  }
}
```

Output from the program:

Click here to view code image

```
    Rounding:   v1           v2
     CEILING:   1.1 ->  2   -1.8 -> -1
       FLOOR:   1.8 ->  1   -1.1 -> -2
          UP:   1.1 ->  2   -1.1 -> -2
        DOWN:   1.8 ->  1   -1.8 -> -1
     HALF_UP:   1.5 ->  2    1.4 ->  1
     HALF_UP:  -1.4 -> -1   -1.5 -> -2
   HALF_DOWN:   1.6 ->  2    1.5 ->  1
   HALF_DOWN:  -1.5 -> -1   -1.6 -> -2
   HALF_EVEN:   2.5 ->  2    1.5 ->  2
   HALF_EVEN:  -1.5 -> -2   -2.5 -> -2
```

Customizing Decimal Number Formatting

The `java.text.DecimalFormat` class formats and parses numbers (including `BigDecimal`) according to a *string pattern*, specified using special *pattern symbols* (Table 18.10) and taking into account any locale considerations. It can format numbers as integers (`1234`), fixed-point notation (`3.14`), scientific notation (`2.99792458E8`), percentages (`60%`), and currency amounts (`$64.00`).

Table 18.10 *Selected Number Format Pattern Symbols for the United States*

Pattern symbol	**Purpose**
`0` (Zero)	Placeholder for a digit. Can result in leading and/or trailing zeroes. Can result in rounding up of the fractional part of a number.
`#` (Hash sign)	Placeholder for a digit. Leading and/or trailing zeroes may be absent. Can result in rounding up of the fractional part of a number.
`.` (Dot)	Placeholder for *decimal separator*. If omitted, a floating-point number is rounded up to the nearest integer. Can be different for other countries. For example, in Germany, the decimal separator is the comma (`,`).
`,` (Comma)	Placeholder for *grouping separator* (a.k.a. *thousands separator*). Can be different for other countries. For example, in Germany, the grouping separator is the period (`.`).

Although the `DecimalFormat` class defines constructors, the Java API recommends using the factory methods of its abstract superclass `NumberFormat` to create an instance of this class.

Click here to view code image

```
NumberFormat df = NumberFormat.getNumberInstance(Locale.US);
```

```
if (nf intanceof DecimalFormat) {
  DecimalFormat df = (DecimalFormat) nf;
  df.applyPattern(pattern);                // Supply the pattern.
  String output = df.format(number);       // Format the number.
}
```

The idiom is to create a `DecimalFormat` object as shown above, localized if necessary, and customizing it with a pattern using the method shown below. The overloaded `format()` methods inherited by the `DecimalFormat` class from its superclass `Number-Format` can now be called to format a number. The pattern can also be changed on the fly.

> Click here to view code image
>
> ```
> void applyPattern(String pattern)
> ```
>
> Applies the given pattern to this `DecimalFormat` object.

Example 18.6 shows formatting of a number, either a `double` at (1a) or a `BigDecimal` at (1b), using different patterns at (2) and different locales at (3). Decimal formatters are created at (4), as explained above. A decimal formatter is customized with a pattern at (5) and the number is formatted at (6). In particular, note how the decimal and group separators are localized for each locale. The currency amounts are also formatted according to the particulars of each locale, and not according to the currency symbol that is specified in the currency patterns.

Example 18.6 *Using the* `DecimalFormat` *class*

Click here to view code image

```
import java.math.BigDecimal;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class FormattingDecimalNumbers {

  public static void main(String[] args) {
    // The number to format.
    double number = 1234.567;                                              // (1
//  BigDecimal number = new BigDecimal("1234.567");                        // (1

    // Formats to use:
    String[] patterns = {                                                  // (2
        "#",
        "###,###.##",
        "###.##",
```

```
        "00000.00",
        "BTC ###,###.##",                                           // BTC: Bitc
    };

    // Locales to consider:
    Locale[] locales = { Locale.US, Locale.GERMANY };                      // (3

    // Create localized DecimalFormats:                                       (4
    List<DecimalFormat> dfs = new ArrayList<>(locales.length);
    for (int i = 0; i < locales.length; i++) {
      NumberFormat nf = NumberFormat.getNumberInstance(locales[i]);
      if (nf instanceof DecimalFormat) {
        dfs.add((DecimalFormat) nf);
      }
    }

    // Write the header:
    System.out.printf("%15s", "Patterns");
    for (Locale locale : locales) {
      System.out.printf("%15s", locale);
    }
    System.out.println();

    // Do formatting and print results:
    for (String pattern : patterns) {
      System.out.printf("%15s", pattern);
      for (DecimalFormat df : dfs) {
        df.applyPattern(pattern);                                        // (5)
        String output = df.format(number);                               // (6)
        System.out.printf("%15s", output);
      }
      System.out.println();
    }
  }
}
```

Output from the program:

Click here to view code image

```
       Patterns          en_US          de_DE
              #           1235           1235
     ###,###.##       1,234.57       1.234,57
         ###.##        1234.57        1234,57
       00000.00       01234.57       01234,57
 BTC ###,###.##   BTC 1,234.57   BTC 1.234,57
```

18.6 Formatting and Parsing Date and Time

The class `java.time.format.DateTimeFormatter` provides the support for locale-sensitive formatting and parsing of *date* and *time* values.

In this section we take a closer look at formatting temporal objects and parsing character sequences for temporal objects. In particular, we consider the following formatters, which provide increasing flexibility in customizing formatting and parsing:

- *Default formatters* are implicitly used by such methods as the `toString()` method of the temporal classes.
- *Predefined formatters* are ready-made formatters provided as constants by the `java.time.format.DateTimeFormatter` class, such as those that adhere to the ISO standard (Table 18.11, p. 1130).

Table 18.11 *Selected ISO-Based Predefined Formatters for Date and Time*

Predefined formatters	Temporal classes	Examples

`ISO_LOCAL_TIME`	`LocalTime`
	Time part of:
	`LocalDateTime`
	`ZonedDateTime`
		`12:30:15`

`BASIC_ISO_DATE`	`LocalDate`
	Date part of:
	`LocalDateTime`
	`ZonedDateTime`
		`20210428`

`ISO_LOCAL_DATE`	`LocalDate`
	Date part of:
	`LocalDateTime`
	`ZonedDateTime`
		`2021-04-28`

`ISO_LOCAL_DATE_TIME`	`LocalDateTime`
	Date/time part of:
	`ZonedDateTime`
		`2021-04-28T12:30:15`

`ISO_ZONED_DATE_TIME`	`LocalDate`
	`LocalTime`
	`ZonedDateTime`
		`2021-04-28T12:30:15+01:00[Europe/Paris]`

`ISO_INSTANT`	`Instant`	`2021-04-28T12:30:15.000000500Z`

- *Style-based formatters* are formatters that use the format styles defined by the constants of the `java.time.format.FormatStyle` enum type (Table 18.12, p. 1132). These formatters are created by the static factory methods `ofLocalized`*`Type`*`()` of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime` (Table 18.13, p. 1132).

Table 18.12 *Format Styles for Date and Time*

Styles	Examples for date/time	Formatting a date (default locale: United States)
`FormatStyle.SHORT`	Short-style pattern	4/28/21
`FormatStyle.MEDIUM`		Jan 11, 2014

Medium-style pattern		
Long-style pattern `FormatStyle.LONG`		
Full-style pattern `FormatStyle.FULL`	`January 11, 2014`	

- *Pattern-based formatters* use customized format styles defined by *pattern letters* (Table 18.15, p. 1135). These formatters are created by the static factory method `ofPattern()` of the `DateTimeFormatter` class.

The `DateTimeFormatter` class provides static factory methods for obtaining a formatter. The idiom for using a formatter is to obtain a formatter first, localize it if necessary, and then pass it to the methods responsible for formatting and parsing temporal objects. Each of the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, and `ZonedDateTime` provides the following methods: an instance method `format()` and a static method `parse()`. These two methods do the formatting and the parsing according to the rules of the formatter that is passed as an argument, respectively.

Analogous methods for formatting and parsing temporal objects passed as an arguments are also provided by the `DateTimeFormatter` class (see below). For example, the following code lines give the same result string, where `date` and `df` refer to a `LocalDate` object and a `DateTimeFormatter` object for formatting dates, respectively:

Click here to view code image

```
String resultStr1 = date.format(df);
String resultStr2 = df.format(date);
boolean eqStr = resultStr1.equals(resultStr2);    // true
```

From the method headers of the `format()` and the `parse()` methods of the temporal classes, we can see that these methods will readily compile with *any* `DateTime-Formatter`. The validity of the formatter for a given temporal object is resolved at runtime, resulting in a resounding exception if it is not valid.

Click here to view code image

```
// Defined by LocalTime, LocalDate, LocalDateTime, and ZonedDateTime
String format(DateTimeFormatter formatter)
```

Formats this temporal object using the specified formatter, and returns the resulting string. Each temporal class provides this method. The temporal object is formatted according to the rules of the formatter. The method throws a

> `java.time.DateTimeException` if formatting is not successful.
>
> Click here to view code image
>
> ```
> static TemporalType parse(CharSequence text)
> static TemporalType parse(CharSequence text, DateTimeFormatter formatter)
> ```
>
> Each temporal class provides these two static methods, where *`TemporalType`* can be any of the temporal classes `LocalTime`, `LocalDate`, `LocalDateTime`, or `Zoned-DateTime`.
>
> The first method returns an instance of the *`TemporalType`* from a character sequence, using the default parsing rules for the *`TemporalType`*.
>
> The second method obtains an instance of the *`TemporalType`* from a character sequence using the specified formatter.
>
> Both methods return an object of a specific temporal class, and both throw a `java.time.format.DateTimeParseException` if parsing is not successful.

Formatters supplied by the `DateTimeFormatter` class are immutable and thread-safe. All formatters created using static methods provided by the `DateTimeFormatter` class can be localized by the `localizedBy()` method in this class. The `DateTimeFormatter` class provides methods that can be used to format and parse temporal objects.

> Click here to view code image
>
> ```
> // Defined by DateTimeFormatter
> DateTimeFormatter localizedBy(Locale locale)
> ```
>
> Returns a copy of this formatter that will use the specified locale (§18.1, p. 1096). Although the formatters in the examples presented in this section use the default locale (the US locale in this case), we encourage the reader to experiment with changing the locale of a formatter with this method in the examples.
>
> Click here to view code image
>
> ```
> String format(TemporalAccessor temporal)
> ```
>
> Formats a temporal object using this formatter. The main temporal classes implement the `TemporalAccessor` interface.
>
> This method throws an unchecked `java.time.DateTimeException` if an error occurs during formatting.

> Click here to view code image
>
> ```
> TemporalAccessor parse(CharSequence text)
> ```
>
> This method fully parses the text to produce a temporal object. It throws an unchecked `java.time.DateTimeException` if an error occurs during parsing.
>
> Click here to view code image
>
> ```
> DateTimeFormatter withZone(ZoneId zone)
> ```
>
> Returns a copy of this formatter with a new override zone—that is, a formatter with similar state to this formatter but with the override zone set.

Default Formatters for Date and Time Values

Default formatters rely on the `toString()` method of the individual temporal classes for creating a text representation of a temporal object. The default formatter used by the `toString()` method applies the formatting rules defined by the ISO standard.

In the following code, the result of formatting a `LocalTime` object is shown at (1):

Click here to view code image

```
LocalTime time = LocalTime.of(12, 30, 15, 99);
String strTime = time.toString();                    // (1) 12:30:15.000000099
LocalTime parsedTime = LocalTime.parse(strTime);  // (2)
System.out.println(time.toString().equals(parsedTime.toString())); // true
```

Each temporal class provides a static method `parse(CharSequence text)` that parses a character sequence using a default formatter that complies with the ISO standard. In the preceding code, the text representation created at (1) is parsed at (2) to obtain a new `LocalTime` object. Not surprisingly, the text representations of the two `LocalTime` objects referred to by the references `time` and `parsedTime` are equal.

The line of code below shows that the argument string passed to the `parse()` method is not in accordance with the ISO standard, resulting in a runtime exception:

Click here to view code image

```
LocalTime badTime = LocalTime.parse("12.30.15");  // DateTimeParseException
```

The examples in this section make heavy use of the `toString()`

method to format temporal objects according to the ISO standard.

Predefined Formatters for Date and Time Values

The `DateTimeFormatter` class provides a myriad of predefined formatters for temporal objects, the majority of which comply with the ISO standard. Table 18.11 shows selected ISO-based predefined formatters from this class. We have also indicated in the last column which temporal classes a formatter can be used with for formatting and parsing.

For example, the row for the `ISO_LOCAL_DATE` formatter in Table 18.11 indicates that this formatter can be used for formatting a `LocalDate` and for formatting the date part of a `LocalDateTime` or a `ZonedDateTime`. It can parse the text representation of a `LocalDate`. In contrast, the row for the `ISO_LOCAL_DATE_TIME` formatter indicates that this formatter can be used for formatting a `LocalDateTime` and for formatting the date-time part of a `ZonedDateTime`. It can parse the text representation of a `LocalDateTime`. In addition, the text representation of a `LocalDateTime` can be parsed by this formatter to a `LocalTime` or a `LocalDate` by the `parse(text, formatter)` method of the appropriate class.

An example of using an ISO-based predefined formatter is given next. Note that the formatter obtained at (1) is a formatter for date fields. It can be used only with temporal objects that have date fields—in other words, the `LocalDate`, `LocalDateTime`, and `ZonedDateTime` classes. This formatter is passed at (2) to the `format()` method, to create a text representation of a date. The resulting string is parsed at (3) by the `parse()` method that uses the same formatter. The resulting date is also formatted using the same formatter at (4). It is hardly surprising that the text representations of both dates are equal.

Click here to view code image

```
DateTimeFormatter df = DateTimeFormatter.ISO_LOCAL_DATE;     // (1)
LocalDate date = LocalDate.of(1935, 1, 8);
String strDate = date.format(df);                             // (2) 1935-01-08
LocalDate parsedDate = LocalDate.parse(strDate, df);          // (3)
System.out.println(strDate + "|" +
                   parsedDate.format(df));         // (4) 1935-01-08|1935-01-
```

As this code shows, a formatter can be reused, both for formatting and for parsing. The code at (4) in the code below applies the formatter from (1) in the preceding code snippet to format a `LocalDateTime` object. It should not come as a surprise that the resulting text representation of the `LocalDateTime` object pertains to only date fields in the temporal object; the time fields of the `LocalDateTime` object are ignored. Parsing this text representation back with the same formatter at (5) will yield only a `LocalDate` object.

Click here to view code image

```
LocalDateTime dateTime = LocalDateTime.of(1935, 1, 8, 12, 45);
String strDate2 = dateTime.format(df);                       // (4) 1935-01-08
LocalDate parsedDate2 = LocalDate.parse(strDate2, df);       // (5) LocalDate
```

To summarize, the `DateTimeFormatter.ISO_LOCAL_DATE` can be used to format and parse a `LocalDate`, but can only format the date part of a `LocalDateTime` object (or a `ZonedDateTime` object).

Using this date formatter with a `LocalTime` object is courting disaster, as shown by the following code. Formatting with this formatter results in an unchecked `java.time.temporal.UnsupportedTemporalTypeException`, and parsing results in an unchecked `java.time.format.DateTimeParseException`.

Click here to view code image

```
String timeStr2 = LocalTime.NOON.format(df);   // UnsupportedTemporalTypeExcep
LocalTime time2 = LocalTime.parse("12:00", df);// DateTimeParseException
```

The `DateTimeFormatter.ISO_INSTANT` predefined formatter shown in Table 18.11 is implicitly used by the `parse(text)` and `toString()` methods of the `Instant` class.

Style-Based Formatters for Date and Time Values

For more flexible formatters than the predefined ISO-based formatters, the `Date-TimeFormatter` class provides the static factory methods `ofLocalized`*`Type`*`()`, where *`Type`* is either `Time`, `Date`, or `DateTime`. These methods create formatters that use a specific format style. However, the format style cannot be changed after the formatter is created. Format styles are defined by the enum type `java.time.format.Format-Style`, and are shown in Table 18.12. The styles define format patterns that vary in their degree of verbosity.

The format style in a style-based formatter can be made locale-specific by setting the desired locale in the formatter using the `localizedBy()` method of the `DateTime-Formatter` class (p. 1128).

Click here to view code image

```
static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)
static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle,
                                             FormatStylse timeStyle)
```

> These static factory methods of the `DateTimeFormatter` class create a formatter that will format a time, a date, or a date-time, respectively, using the specified format style. The formatter can also be used to parse a character sequence for a time, a date, or a date-time, respectively.

In the discussion below, we will make use of the following temporal objects.

Click here to view code image

```
LocalTime time = LocalTime.of(14, 15, 30);               // 14:15:30
LocalDate date = LocalDate.of(2021, 12, 1);              // 2021-12-01
LocalDateTime dateTime = LocalDateTime.of(date, time); // 2021-12-01T14:15:30
// 2021-12-01T14:15:30-06:00[US/Central]
ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, ZoneId.of("US/Central
```

Table 18.13 shows which temporal classes can be formatted by a combination of a format style from Table 18.12 and an `ofLocalized`*`Type`*`()` method of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime`. For example, in Table 18.13 we can see that the method call `DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)` will return a formatter that can be used to format instances of the `LocalDate` class, but will only format the date part of a `LocalDateTime` or a `ZonedDateTime` instance, as can be seen in the code below. This particular formatter requires date fields in the temporal object, and instances of these three temporal classes fit the bill.

Table 18.13 *Using Style-Based Formatters*

Using formatters created by factory methods of the `DateTimeFormatter` class: *`temporalReference`*`.format(formatter)`			
`FormatStyle`	**`ofLocalizedTime(style)`**	**`ofLocalizedDate(style)`**	**`ofLocalizedDateTime(style)`**

`SHORT`
`MEDIUM`
	`LocalTime`
	Time part of:
	`LocalDateTime`
	`ZonedDateTime`
		`LocalDate`
		Date part of:
		`LocalDateTime`
		`ZonedDateTime`
			`LocalDateTime`
			`ZonedDateTime`

`LONG`
`FULL`
	Time part of:
	`ZonedDateTime`
		`LocalDate`
		Date part of:
		`LocalDateTime`
		`ZonedDateTime`
			`ZonedDateTime`

Click here to view code image

```
DateTimeFormatter dfs = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
String str1 = date.format(dfs);          // 12/1/21
String str2 = dateTime.format(dfs);      // Date part: 12/1/21
String str3 = zonedDateTime.format(dfs); // Date part: 12/1/21
```

Similarly, the method call `DateTimeFormatter.ofLocalizedTime(FormatStyle.LONG)` will return a formatter that will only format temporal objects that have a time part *and* are compatible with the rules of `FormatStyle.LONG`. Only instances of the `Zoned-DateTime` class fit the bill. The code below shows that we can format the time part of a `ZonedDateTime` instance with this formatter.

Click here to view code image

```
DateTimeFormatter tff = DateTimeFormatter.ofLocalizedTime(FormatStyle.FULL);
String str4 = zonedDateTime.format(tff); // Time part:
                                         //   2:15:30 PM Central Standard Tim
String str5 = time.format(tff);          // java.time.DateTimeException
String str6 = date.format(tff);          // java.time.temporal.
                                         //      UnsupportedTemporalTypeExcep
String str7 = dateTime.format(tff);      // java.time.DateTimeException
```

In summary, a style-based formatter will only format a temporal object (or its constituent parts) if the temporal object has the temporal parts required by the formatter *and* is compatible with the rules of the format style of the formatter.

Table 18.14 shows how the style-based formatters can be used as parsers. As in Table 18.13, this table also shows the combination of a format style from Table 18.12 and an `ofLocalized`*`Type`*`()` method of the `DateTimeFormatter` class, where *`Type`* is either `Time`, `Date`, or `DateTime`. From the table, we can read which temporal objects can be parsed from a character sequence that is compatible with the rules of a formatter for a given combination of the format style and a specific factory method.

Table 18.14 *Using Style-Based Parsers*

Using parsers created by factory methods of the `DateTimeFormatter` class: ***`TemporalClass`*`.parse(characterSequence,formatter)`**			
`Format Style`	**`ofLocalizedTime(style)`**	**`ofLocalizedDate(style)`**	**`ofLocalizedDateTime(style)`**
`SHORT` `MEDIUM`	`LocalTime`	`LocalDate`	`LocalDateTime`
`LONG` `FULL`	`LocalTime`	`LocalDate`	`LocalDateTime`

`ZonedDateTime`

For any format style, the two methods `ofLocalizedTime()` and `ofLocalizedDate()` return formatters that can be used to parse a compatible character sequence to a `LocalTime` or a `LocalDate` instance, respectively.

In the code that follows, the date formatter created at (1) is used at (3) to parse the input string at (2) to create a `LocalDate` object.

Click here to view code image

```
DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);//
String inputStr = "2/29/21";                       // (2) en_US date, SHORT st
LocalDate parsedDate = LocalDate.parse(inputStr, df);    // (3)
System.out.println(parsedDate);                          // (4) 2021-02-28
```

In the above code, the input string `"2/29/21"` is specified in the short style of the default locale (which in our case is the US locale). The input string is parsed by the date formatter (using the `SHORT` format style) to create a new `LocalDate` object.

Although the value 29 is invalid for the number of days in February for the year 2021, the output shows that it was adjusted correctly. The contents of the input string (in this case, `"2/29/21"`) must be compatible with the rules of the format style in the date formatter (in this case, `FormatStyle.SHORT`). If this is not the case, a `DateTimeParse-Exception` is thrown. Note that in the print statement at (4), the `LocalDate` object from the parsing is converted to a string by the `LocalDate.toString()` method using the implicit ISO-based formatter.

A formatter returned by the `ofLocalizedDateTime()` method can parse a character sequence to instances of different temporal classes, as shown in the rightmost column of Table 18.14. This is illustrated by the code below. Such a formatter is created at (1) and localized to the locale for France. It is first used to format a zoned date-time object at (2) to obtain a character string that we can parse with the formatter. At (3), (4), (5), and (6), this sequence is parsed to obtain a `LocalTime`, a `Local-Date`, a `LocalDateTime`, and a `ZonedDateTime`, respectively, as these temporal objects can be constructed from the format of the character string representing a zoned date-time.

Click here to view code image

```
DateTimeFormatter dtff
    = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL)
                       .localizedBy(Locale.FRANCE);          // (1)
```

```
// "mercredi 1 décembre 2021 à 14:15:30 heure normale du centre nord-américain
String charSeq = zonedDateTime.format(dtff);                    // (2)
LocalTime pTime = LocalTime.parse(charSeq, dtff);               // (3) 14:15:30
LocalDate pDate = LocalDate.parse(charSeq, dtff);               // (4) 2021-12-01

// 2021-12-01T14:15:30
LocalDateTime pDateTime = LocalDateTime.parse(charSeq, dtff);          // (5)

// 2021-12-01T14:15:30-06:00[America/Chicago]
ZonedDateTime pZonedDateTime = ZonedDateTime.parse(charSeq, dtff);    // (6)
```

In summary, a character sequence can be parsed to a temporal object by a style-based formatter if the character sequence is in the format required by the formatter to create the temporal object.

Pattern-Based Formatters for Date and Time Values

For more fine-grained formatting and parsing capabilities for temporal objects, we can use the `ofPattern()` method of the `DateTimeFormatter` class. This method creates immutable formatters that interpret temporal objects according to a string pattern that is defined using the *pattern letters* shown in Table 18.15.

Table 18.15 *Selected Date/Time Pattern Letters*

Date or time component	Pattern letter	Example

Year
(2: two rightmost digits) Proleptic year: use `u` Year of era (AD/BC): use `y`
	`u`
	`uu`
	`uuu`
	`uuuu`
	`uuuuu`
		`2021; -2000 (2001 BC)`
		`15;    0 (i.e., 1 BC); -1 (i.e., 2 BC)`
		`2021;  2021 (i.e., 2022 BC)`
		`02021 (padding)`

Month in year
(1–2: number)
(3: abbreviated text form)
(4: full text form)
	`M`
	`MM`
	`MMM`
	`MMMM`
		`8`
		`08`
		`Aug`
		`August`

Day in month	`d` `dd`	`6` `06`

Day name in week
(1–3: abbreviated text form)
(4: full text form)
	`E`
	`EE`
	`EEE`
	`EEEE`
		`Tue`
		`Tue`
		`Tue`
		`Tuesday`

Hour in day (0–23)	`H` `HH`	`9` `09`
Hour in am/pm (1–12) (does not include the AM/PM marker, but required for parsing)	`h` `hh`	`7` `07`
Minute in hour (0–59)	`m` `mm`	`6` `06`
Second in minute (0–59)	`s` `ss`	`2` `02`

Fraction of a second (`S`)	`S`
	`SSS`
	`SSSSSS`
	`SSSSSSSSS`
		`1`
		`123`
		`123456`
		`123456789`

	`G`	`AD`

Era designator (AD/BC)	
Time zone name (1 to 4)	`CST` `Central Standard Time`

Time zone offset (1 to 5)	`-0600`
	`GMT-06:00`
	`-06:00`

Time zone ID (must be two)	`US/Central`
AM/PM marker	`AM`

Period-of-day (used with temporal values having time units)	`at night`
	`in the morning`
	`noon`
	`in the afternoon`
	`in the evening`

Escape for text	`prints as`
Single quote	`'`

> [Click here to view code image]
>
> ```
> static DateTimeFormatter ofPattern(String pattern)
> ```
>
> This static method creates a formatter using the specified pattern. The set of temporal objects it can be used with depends on the pattern letters used in the specification of the pattern. The letter pattern defines the rules used by the formatter. The method throws an `IllegalArgumentException` if the pattern is invalid.

Table 18.15 provides an overview of selected pattern letters. All letters are reserved when used in a letter pattern. A sequence of characters can be escaped by enclosing it in single quotes (e.g., `"EEEE 'at' HH:mm"`). Non-letter characters in the string are interpreted verbatim and need not be escaped using single quotes (e.g., `"uuuu.MM.dd @ HH:mm:ss"`). The number of times a pattern letter is repeated can have a bearing on the interpretation of the value of the corresponding date or time field. The uppercase letter `M` (Month of the year) should not be confused with the lowercase letter `m` (minutes in the hour).

A letter pattern can be used to format a temporal object if the temporal object has the temporal fields required by the pattern. The pattern `"'Hour': HH"` can be used to format the hour part of any `LocalTime` object or a `LocalDateTime` object, but not a `LocalDate`.

A letter pattern can be used to parse a string if the string matches the pattern *and* the letter pattern specifies the mandatory parts needed to construct a temporal object. The pattern `"MM/dd/uuuu"` can be used to parse the string `"08/13/2009"` to obtain a `LocalDate` object, but not a `LocalDateTime` object. The latter requires the time part as well.

Example 18.7 demonstrates both formatting temporal objects and parsing character sequences for temporal objects using letter patterns. The `main()` method at (1) calls four methods to format and parse different temporal objects. The formatting of the temporal object can be localized by specifying the desired locale in the `main()` method.

- The method `usingTimePattern()` at (2) demonstrates using a letter pattern for the time part to both format a `LocalTime` and parse a text representation of a `LocalTime`, respectively. The same pattern is used to format only the time part of a `Local-DateTime` and a `ZonedDateTime`, respectively.
- The method `usingDatePattern()` at (3) demonstrates using a letter pattern for the date part to both format a `LocalDate` and parse a text representation of a `Local-Date`, respectively. The same pattern is used to format only the date part of a `LocalDateTime` and a `ZonedDateTime`, respectively.
- The method `usingDateTimePattern()` at (4) demonstrates using a letter pattern for the date and time parts to both format a `LocalDateTime` and parse a text representation of a `LocalDateTime`, respectively. The same pattern is also used to parse the text representation of a `LocalDateTime` to obtain a `LocalDate` and a `LocalTime`, respectively.
- The method `usingZonedDateTimePattern()` at (5) demonstrates using a letter pattern for the date, time, and zone parts to both format a `ZonedDateTime` and parse a text representation of a `ZonedDateTime`, respectively. The same pattern is also used to parse the text representation of a `ZonedDateTime` to obtain a `LocalDateTime`, a `LocalDate`, and a `LocalTime`, respectively.

The usage of letter patterns with the `ofPattern()` method in Example 18.7 is analogous to the usage of style-based formatters provided by the `ofLocalized`*`Type`*`()` methods (Table 18.13, p. 1132). The main difference is that letter patterns provide great flexibility in creating customized format styles.

Example 18.7 *Formatting and Parsing with Letter Patterns*

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;

public class FormattingParsingWithPatterns {

  /** Temporals */
  private static LocalTime time = LocalTime.of(12, 30, 15, 99);
  private static LocalDate date = LocalDate.of(2021, 4, 28);
  private static LocalDateTime dateTime = LocalDateTime.of(date, time);
  private static ZoneId zID = ZoneId.of("US/Central");
  private static ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, zID)

  public static void main(String[] args) {                                  //
    Locale locale = Locale.US;
    usingTimePattern(locale);
    usingDatePattern(locale);
    usingDateTimePattern(locale);
    usingZonedDateTimePattern(locale);
  }

  /** Pattern with time part. */
  public static void usingTimePattern(Locale locale) {                      //
    String timePattern = "HH::mm::ss:SSS";
    DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern(timePattern)
                                                       .localizedBy(locale);
    String strTime = time.format(timeFormatter);
    LocalTime parsedTime = LocalTime.parse(strTime, timeFormatter);
    String strTime2 = dateTime.format(timeFormatter);
    String strTime3 = zonedDateTime.format(timeFormatter);

    System.out.printf("Time pattern: %s%n", timePattern);
    System.out.printf("LocalTime (formatted): %s%n", strTime);
    System.out.printf("LocalTime (parsed):    %s%n", parsedTime);
    System.out.printf("LocalDateTime (formatted time part): %s%n", strTime2);
    System.out.printf("ZonedDateTime (formatted time part): %s%n%n", strTime3)
  }

  /** Pattern with date part. */
  public static void usingDatePattern(Locale locale) {                      //
    String datePattern = "EEEE, uuuu/MMMM/dd";
    DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern(datePattern)
                                                       .localizedBy(locale);
    String strDate = date.format(dateFormatter);
    LocalDate parsedDate = LocalDate.parse(strDate, dateFormatter);
    String strDate2 = dateTime.format(dateFormatter);
    String strDate3 = zonedDateTime.format(dateFormatter);

    System.out.printf("Date pattern: %s%n", datePattern);
    System.out.printf("LocalDate (formatted): %s%n", strDate);
    System.out.printf("LocalDate (parsed)   : %s%n", parsedDate);
    System.out.printf("LocalDateTime (formatted date part): %s%n", strDate2);
    System.out.printf("ZonedDateTime (formatted date part): %s%n%n", strDate3)
  }

  /** Pattern with date and time parts. */
  public static void usingDateTimePattern(Locale locale) {                  //
    String dtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd";
    DateTimeFormatter dtFormatter = DateTimeFormatter.ofPattern(dtPattern)
                                                     .localizedBy(locale);
    String strDateTime = dateTime.format(dtFormatter);
    LocalDateTime parsedDateTime = LocalDateTime.parse(strDateTime,
                                                       dtFormatter);
    LocalDate parsedDate3 = LocalDate.parse(strDateTime, dtFormatter);
    LocalTime parsedTime3 = LocalTime.parse(strDateTime, dtFormatter);

    System.out.printf("DateTime pattern: %s%n", dtPattern);
    System.out.printf("LocalDateTime (formatted):    %s%n", strDateTime);
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate3);
    System.out.printf("LocalTime (parsed time part): %s%n%n", parsedTime3);
  }

  /** Pattern with time zone, date and time parts. */
  public static void usingZonedDateTimePattern(Locale locale) {              //
    String zdtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd VV";
    DateTimeFormatter zdtFormatter = DateTimeFormatter.ofPattern(zdtPattern)
                                                      .localizedBy(locale);
    String strZonedDateTime = zonedDateTime.format(zdtFormatter);
    ZonedDateTime parsedZonedDateTime
        = ZonedDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDateTime parsedDateTime2
        = LocalDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDate parsedDate4 = LocalDate.parse(strZonedDateTime,
                                            zdtFormatter);
    LocalTime parsedTime4 = LocalTime.parse(strZonedDateTime,
                                            zdtFormatter);

    System.out.printf("ZonedDateTime pattern: %s%n", zdtPattern);
    System.out.printf("ZonedDateTime (formatted):    %s%n", strZonedDateTime);
    System.out.printf("ZonedDateTime (parsed):       %s%n", parsedZonedDateTim
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime2);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate4);
    System.out.printf("LocalTime (parsed time part): %s%n", parsedTime4);
  }
}
```

Probable output from the program:

Click here to view code image

```
Time pattern: HH::mm::ss:SSS
LocalTime (formatted): 12::30::15:000
LocalTime (parsed):    12:30:15
LocalDateTime (formatted time part): 12::30::15:000
ZonedDateTime (formatted time part): 12::30::15:000

Date pattern: EEEE, uuuu/MMMM/dd
LocalDate (formatted): Wednesday, 2021/April/28
LocalDate (parsed)   : 2021-04-28
LocalDateTime (formatted date part): Wednesday, 2021/April/28
ZonedDateTime (formatted date part): Wednesday, 2021/April/28

DateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd
LocalDateTime (formatted):    Wed, 12::30::15 on 2021/04/28
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15

ZonedDateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd VV
ZonedDateTime (formatted):    Wed, 12::30::15 on 2021/04/28 US/Central
ZonedDateTime (parsed):       2021-04-28T12:30:15-05:00[US/Central]
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15
```

18.7 Formatting and Parsing Messages

A *compound message* may contain various kinds of data: strings, numbers, currencies, percentages, dates, and times. The locale-sensitive values in such messages should be formatted according to an appropriate locale. In this section we first consider a solution provided by the `java.text.MessageFormat` class for formatting compound messages. Later we consider the `java.text.ChoiceFormat` class that supports conditional formatting (p. 1145).

An overview of various other support in the Java SE APIs for formatting compound messages can be found at the end of this section (p. 1152).

Format Patterns

The modus operandi of formatters provided by the `MessageFormat` class is a classic one, as exemplified by the `printf()` method called on the `System.out` static field (§1.9, p. 24). A *string pattern* designating *placeholders* for values is specified. `MessageFormat` takes a list of values and formats them, and then inserts their text representation into the respective placeholders in the pattern to create the final result. The simple example below illustrates the basic use of the `MessageFormat` class.

Click here to view code image

```
String pattern = "At {3} on {2} Elvis landed at {0} and was greeted by {1} fan
String output = MessageFormat.format(pattern, "Honolulu", 3000,
                              LocalDate.of(1961,3,25), LocalTime.of(12,15));
System.out.println(output);
```

Output from the code:

Click here to view code image

```
At 12:15 on 1961-03-25 Elvis landed at Honolulu and was greeted by 3,000 fans.
```

The pattern specifies placeholders (called *format elements*) using curly brackets `{}` to designate where text representations of values are to be inserted. The number specified in a format element is the *argument index* of a particular argument in the list of arguments submitted to the formatter. In the code above, the pattern has four format elements.

The variable arity static method `MessageFormat.format()` is passed the pattern and a list of arguments to format. Below, we can see which argument in the method call is indicated by the argument index in a format element that is specified in the pattern above.

Click here to view code image

```
Format element:     {0}        {1}             {2}                       {3}
Arguments:        "Honolulu", 3000, LocalDate.of(1961,3,25), LocalTime.of(12,15
```

Care must be taken that an argument index in a format element designates the right argument. The same argument index in multiple format elements just means that the argument it designates is applied to all of those format elements.

Note that the text outside of the format elements is copied verbatim to the result string. A single quote (`'`) can be used to quote arbitrary characters, and two single quotes (`''`) can be used to escape a single quote in a pattern. For example, the pattern `"'{1}'"` represents the string `"{1}"`, and not a format element.

The flexibility of the `MessageFormat` class will become clear as we dig into its functionality.

The general syntax of a format element is shown below:

Click here to view code image

```
{ Argument_Index }
{ Argument_Index, Format_Type }
{ Argument_Index, Format_Type, Format_Style }
```

The format type and the format style allow further control over the formatting. The *format type* indicates what kind of argument (e.g., `number`, `date`, or `time`) is to be formatted, and the *format style* indicates how the argument will be formatted (e.g., `currency` format for a `number`, `short` format for a `date`). Legal combinations of format type and format style are shown in Table 18.16.

Table 18.16 *Format Type and Format Style Combinations for Format Elements*

Format type	**Format style value that can be specified for a format type**
none	
`number`	*none*, `integer`, `currency`, `percent`, *subformat pattern*
`date` or `time`	*none*, `short`, `medium`, `long`, `full`, *subformat pattern*
`choice` (p. 1145)	*subformat pattern*

The first variant of the format element (first row in Table 18.16) was used in the code earlier, where neither format type nor format style is specified. In this case, the text representation of the argument as defined by the `toString()` method is used in the pattern to create the final result. We will explore other combinations from Table 18.16 in this section.

When format type and format style values are used in a format element, an appropriate formatter is implicitly created to format the argument corresponding to that particular format element. The formatters created are instances of the formatter classes shown in Figure 18.1, p. 1115. For example, the format element `{3,time,short}` results in a locale-specific formatter being created implicitly by the following method call:

Click here to view code image

```
DateFormat.getTimeInstance(DateFormat.SHORT, getLocale())
```

This formatter will format a `java.util.Date` object (which represents both date and time) by extracting its time components and formatting them according to the format defined by `DateFormat.SHORT` and taking the locale into consideration. If a `MessageFormat` instance does not specify the locale, then the default locale is used.

Format elements that specify the format types `date` and `time` are compatible with the `java.util.Date` legacy class, with subsequent reliance on the `DateFormat` class to provide an appropriate formatter. This presents a slight problem when we want to format date and time values of the new Date and Time API. We will use the static methods of the utility class `ConvertToLegacyDate` (§17.8, p. 1088) to convert `Local-Date`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime` objects to `Date` objects in order to leverage the format element handling functionality of the `MessageFormat` class.

The pattern that we saw earlier with just vanilla format elements (i.e., format elements with just the argument index) is now shown at (1) below to include the specification of the type and style of the format elements. A `MessageFormat` instance that is based on the pattern at (1) is created at (2) using a constructor of the `MessageFormat` class. The arguments that we want to format are included in the `Object` array at (3). Note that the element index in this array corresponds to an argument index in the format elements specified in the pattern. We use the methods from the utility class `ConvertToDate` to convert `LocalTime` and `LocalDate` values to `Date` values (§17.8, p. 1088). Finally, at (4), the argument array is passed for formatting to the `format()` method inherited by the `MessageFormat` class from its superclass `Format`. The code below also outlines the pertinent steps that are involved when using a `Message-Format` instance to format compound messages.

Click here to view code image

```
// Specify the pattern:
String pattern2 = "At {3,time,short} on {2,date,medium} Elvis landed at {0} "
                + "and was greeted by {1,number,integer} fans.";

// Create a MessageFormat based on the given pattern:
MessageFormat mf2 = new MessageFormat(pattern2);

// Create the array with the arguments to format:
Object[] messageArguments = {
    "Honolulu",                                          // argument index 0
    3000,                                                // argument index 1
    ConvertToDate.ldToDate(LocalDate.of(1961,3,25)),     // argument index 2
    ConvertToDate.ltToDate(LocalTime.of(12,15))          // argument index 3
};

// Format the arguments:
String output2 = mf2.format(messageArguments);
System.out.println(output2);
```

Output from the code:

Click here to view code image

```
At 12:15 PM on Mar 25, 1961 Elvis landed at Honolulu and was greeted by 3,000
```

This output is different from the output we saw earlier when using vanilla format elements. In this case, appropriate formatters are implicitly created for each format element that apply the format style and take into consideration the locale (in this case, it is the default locale, which happens to be the US locale).

The following constructors defined by the `MessageFormat` class are used in the examples in this section:

Click here to view code image

```
MessageFormat(String pattern)
MessageFormat(String pattern, Locale locale)
```

Create a message formatter to apply the specified pattern. The created formatters will format or parse according to the default locale or the specified locale, respectively.

Selected methods are provided by the `MessageFormat` class, many of which are used in the examples in this section:

Click here to view code image

```
final String format(Object obj)       // Inherited from the Format class.
```

Returns a string that is the result of formatting the specified object. Passing an `Object[]` formats the elements of the array individually.

Click here to view code image

```
static String format(String pattern, Object... arguments)
```

This static method implicitly creates a one-time formatter with the given pattern, and returns the result of using it to format the variable arity arguments.

```
void setLocale(Locale locale)
```

Sets the locale to be used when creating subformats. Subformats already created are not affected.

```
Locale getLocale()
```

Returns the locale used by this formatter.

```
void applyPattern(String pattern)
```

Sets the pattern used by this formatter by parsing the pattern and creating the necessary subformats.

```
String toPattern()
```

Returns a pattern that represents the current state of this formatter.

Click here to view code image

```
void setFormat(int formatElementIndex, Format newFormat)
void setFormats(Format[] newFormats)
```

The first method sets the format to use for the format element indicated by the `formatElementIndex` in the previously set pattern string.

The second method sets the formats to use for the format elements in the previously set pattern string. Note that the order of formats in the `newFormats` array corresponds to *the order of format elements in the pattern string*. Contrast this method with the `setFormatsByArgumentIndex()` method.

More formats provided than needed by the pattern string are ignored. Fewer formats provided than needed by the pattern string results in only the provided formats being inserted.

Click here to view code image

```
void setFormatByArgumentIndex(int argumentIndex, Format newFormat)
void setFormatsByArgumentIndex(Format[] newFormats)
```

> The first method sets the format to use for the format element indicated by the `argumentIndex` in the previously set pattern string.
>
> The second method sets the formats to use for the format elements in the previously set pattern string. However, note that the order of formats in the `new-Formats` array corresponds to *the order of elements in the arguments array* passed to the format methods or returned by the parse methods. Contrast this method with the `setFormats()` method.
>
> Any argument index not used for any format element in the pattern string results in the corresponding new format being ignored.
>
> Fewer formats being provided than needed results in only the formats for provided argument indices being replaced.

Formatting Compound Messages

If the application is intended for an international audience, we need to take the locale into consideration when formatting compound messages.

Example 18.8 illustrates formatting compound messages for different locales, where locale-sensitive data is contained in resource bundles (p. 1102). The program output shows stock information about an item according to the requested locale, showing how formatting of number, currency, date, and time is localized. A resource bundle file is created for each locale. The resource bundle for the US locale is shown in Example 18.8. The program output shows stock information for the US locale and the locale for Spain.

Example 18.8 *Formatting Compound Messages*

Click here to view code image

```
# File: StockInfoBundle.properties
pattern = Stock date: {3,time,short}, {4,date,long}\n\
          Item name: {0}\n\
          Item price: {1,number,currency}\n\
          Number of items: {2,number,integer}
item.name = Frozen pizza
item.price = 9.99
```

Click here to view code image

```
import java.text.*;
import java.time.*;
import java.util.*;

public class CompoundMessageFormatting {
  static void displayStockInfo(Locale requestedLocale) {                  // (
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the relevant resource bundle:                                   (
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.StockInfoBundle", requestedLocale)

    // Create a formatter, given the pattern and the locale:                 (
    MessageFormat mf = new MessageFormat(bundle.getString("pattern"),
                                                requestedLocale);

    // Argument values:                                                      (
    String itemName = bundle.getString("item.name");
    double itemPrice = Double.parseDouble(bundle.getString("item.price"));
    int numOfItems = 1234;
    Date timeOnly = ConvertToLegacyDate.ltToDate(LocalTime.of(14,30));
    Date dateOnly = ConvertToLegacyDate.ldToDate(LocalDate.of(2021,3,1));

    // Create argument array:                                                (
    Object[] messageArguments = {
        itemName,       // {0}
        itemPrice,      // {1,number,currency}
        numOfItems,     // {2,number,integer}
        timeOnly,       // {3,time,short}
        dateOnly,       // {4,date,long}
    };

    // Apply the formatter to the arguments:                                 (
    String result = mf.format(messageArguments);
    System.out.println(result);
  }

  public static void main(String[] args) {
    displayStockInfo(Locale.US);
    System.out.println();
    displayStockInfo(new Locale("es", "ES"));
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Requested Locale: en_US
Stock date: 2:30 PM, March 1, 2021
Item name: Frozen pizza
Item price: $9.99
Number of items: 1,234

Requested Locale: es_ES
Fecha de stock: 14:30, 1 de marzo de 2021
Nombre del árticulo: Pizza congelada
Precio del articulo: 8,99 ?
Número de artículos: 1.234
```

In Example 18.8, the method `displayStockInfo()` at (1) epitomizes the basic steps for using `MessageFormat` for formatting compound messages. The method is passed the requested locale for which the stock information should be displayed. It goes without saying that the stock information is printed only if the relevant resource bundle files can be found.

The `MessageFormat` instance created at (2) is locale-specific. The pattern is read from the resource bundle. It is composed of several lines, and appropriate format elements for each kind of value in the pattern are specified, as shown in the resource bundle for the US locale. The `MessageFormat` instance implicitly creates the necessary formatters for type and style specified in the format elements, and applies them to the corresponding arguments.

The arguments to format are set up at (4). We convert `LocalTime` and `LocalDate` instances to `Date` instances. Another solution is to convert these instances to their text representation beforehand using an appropriate `DateTimeFormatter`. It is also possible to set specific formatters for any argument that is to be formatted. We leave the curious reader to explore these solutions for formatting compound messages.

At (5), an `Object` array is initialized with the arguments. Finally, at (6), the `format()` method is invoked on the formatter, passing the argument array. Again note that the currency delimiter for `?` (euro) is a nbsp in the formatted output, as discussed earlier (p. 1116).

Keep in mind that a `MessageFormat` can be reused, with a new locale, a new pattern, and new arguments:

Click here to view code image

```
MessageFormat mf = new MessageFormat(previousPattern);
mf.setLocale(newLocale);
mf.applyPattern(newPattern);
String output = mf.format(newMessageArguments);
```

Conditional Formatting

Consider *contiguous half-open intervals* on the number line, as shown below. The (lower and upper) *limits* of these intervals can be any numbers, which will always be in ascending order as they are on the number line. Below we see two half-open intervals with limits 0.0, 1.0, and 2.0.

If a value v falls in the half-open interval $[v_i, v_{i+1})$, then we select the (lower) limit v_i. So, if v is any number in the half-open interval `[0.0, 1.0)`, the limit 0.0 is selected. Similarly, if v is any number in the half-open interval `[1.0, 2.0)`, the limit 1.0 will be selected, and so on.

From Table 18.16, we see that the format element having the format type `choice` requires a *subformat pattern*. This subformat pattern specifies a *choice format*. The general syntax of the choice format is shown below, where limit v_i and its associated subformat pattern f_i are separated by the hash sign (#). Each $v_i\#f_i$ constitutes *a choice*, and choices are separated by the vertical bar (|). Two consecutive limits constitute a half-open interval: $[v_i, v_{i+1})$. Note that the last choice uses `<`, meaning any value greater than or equal to v_n. Also, the limits $v_1, v_2, \ldots, v_n$ must be in ascending order for the choice format to work correctly.

$$v_1\#f_1 \mid v_2\#f_2 \mid \ldots \mid v_n<f_n$$

The choice format thus defines the limits and the corresponding choices. When a value is supplied for formatting, the choice format determines the limit, as explained above, and the formatter uses the choice associated with this limit— hence the term *conditional formatting*.

Below is a pattern that uses a choice format. The pattern's only format element has *argument index* 0 and *format type* `choice`, and its subformat pattern specifies a *choice format* (underlined below, but see also Table 18.16). The argument index `0` in the pattern designates the value that is used to determine the limit, and thereby, select the corresponding choice in the choice format.

Click here to view code image

```
"There {0,choice,0#are no bananas|1#is only one banana|\
                                    2<are {1,number,integer} bananas}."
```

The choice format above has three choices, separated by the vertical bar (|). Its limits are 0, 1, and 2. Each limit in a choice is associated with a subformat pattern. For limits 0 and 1, the subformat pattern is just text. The subformat pattern for limit 2 specifies a format element to format a number as an integer, as it expects an argument to be supplied for this format element. Note that the argument index in this embedded format element is 1.

Example 18.9 uses a pattern that includes the choice format above to illustrate handling of plurals in messages. Resource bundles for different locales contain the pattern with the choice format. An appropriate resource bundle for a locale is fetched at (1). The choice pattern is read from the resource bundle and a

`MessageFormat` is created at (2), based on the choice pattern and the requested locale. For each locale, the formatter is tested for three cases, exemplified by the rows of two-dimensional array `messageArguments` at (3). Each row of arguments is passed to the `format()` method at (5) for formatting. The program output shows the arguments passed and the corresponding result.

In the argument matrix, the first row, `{0.5}`, specifies the argument `0.5`, which falls in the half-interval `[0, 1)`. Limit `0` is chosen, whose choice is selected. This results in the following pattern to be applied:

```
"There are no bananas."
```

Similarly for the second row, `{1.5}`, the limit is determined to be `1`, resulting in the following pattern to be applied:

```
"There is only one banana."
```

For the third row, `{2.5, 2}`, the argument `2.5` determines the limit `2`, resulting in the pattern below to be applied. The second element (value `2`) in this row is the argument that is formatted according to the format element in this pattern.

Click here to view code image

```
"There are {1,number,integer} bananas."
```

We see from the program output that the correct singular or plural form is selected depending on the value of the argument passed to the formatter. It is instructive to experiment with passing different values to the formatter.

Example 18.9 *Using Choice Pattern*

Click here to view code image

```
# File: ChoiceBundle.properties
choice.pattern = There {0,choice,0#are no bananas|1#is only one banana|\
                                 2<are {1,number,integer} bananas}.
...
```

Click here to view code image

```
import java.text.*;
import java.util.*;

public class ChoicePatternUsage {

  static void displayMessages(Locale requestedLocale) {
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the resource bundle:
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // choice.pattern = There {0,choice,0#are no bananas|1#is only one banana
    //                                  |2#are {1,number,integer} bananas}.
    // Create formatter for specified choice pattern and locale:
    MessageFormat mf =
        new MessageFormat(bundle.getString("choice.pattern"), requestedLocale)

    // Create the message argument arrays:
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);            //
      System.out.printf("Arguments:%-10sResult: %s%n",
                 Arrays.toString(messageArguments[choiceNumber]),result);
    }
  }

  public static void main(String[] args) {
    displayMessages(new Locale("en", "US"));
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

Output from the program:

Click here to view code image

```
Requested Locale: en_US
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested Locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

Example 18.9 illustrates using a choice format for handling plurals. Example 18.10 illustrates an alternative approach where the choice format is created programmatically using the `ChoiceFormat` class.

The class `ChoiceFormat` provides the following constructors for creating a choice format either programmatically or using a pattern.

> Click here to view code image
>
> ```
> ChoiceFormat(double[] limits, String[] formats)
> ```
>
> Creates a `ChoiceFormat` with the limits specified in ascending order and the corresponding formats.
>
> Click here to view code image

> `ChoiceFormat(String newPattern)`
>
> Creates a `ChoiceFormat` with limits and corresponding formats based on the specified pattern string.

The `ChoiceFormat` class is locale-agnostic—that is, providing no locale-specific operations. However, the class `MessageFormat` implements locale-specific operations, and allows formats to be set for individual format elements in its pattern. By setting a `ChoiceFormat` as a format for a choice format element in a locale-sensitive `MessageFormat`, it is possible to achieve locale-specific formatting behavior in a `ChoiceFormat`. This approach is illustrated in Example 18.10. The numbers below correspond to the numbers on the lines in the source code for Example 18.10.

(1) The locale-specific resource bundle with resources for the pattern and for constructing the choice format is fetched. See the listing of the resource bundle file in Example 18.10.

(2) A locale-specific `MessageFormat` is created based on the pattern (`"There {0}."`) which is read from the resource bundle.

(3) To create a `ChoiceFormat` programmatically requires two equal-length arrays that contain the limits and the corresponding subformats, respectively. The `double` array `limits` is created with the limit values `0`, `1`, and `2`. A `String` array `messageFormats` is created by reading the values of the keys `none`, `singular`, and `plural` from the resource bundle. These are the subformats corresponding to the limits that we saw in the choice format in Example 18.9.

(4) Given the arrays for the limits and the subformats, the `ChoiceFormat` constructor creates the choice format.

(5) and (6) One way to set customized formats for individual format elements in a `MessageFormat` is by first creating an array of `Format` (superclass of format classes in `java.text` package). The order of the formats in this array must be the same as either *the order of format elements in the pattern string* or *the order of elements in the argument array passed to the format methods*. There is only one format element in the pattern (`"There {0}."`). The array of `Format` thus has only one element which is the

choice format for this format element, and therefore, both orders are valid in this case. We can use either the `setFormats()` or the `setFormatsByArgumentIndex()` method, as shown at (6a) and (6b), respectively. Either method will associate the choice format with the format element having the argument index 0 in the pattern.

However, the format for the format element at the argument index 0 in the pattern can easily be set by calling the `setFormat()` method, as shown at (6c).

(7) through (9) The behavior of the program is exactly the same as for Example 18.9. Experimenting with other limits and arguments is recommended.

Example 18.10 *Using the* `ChoiceFormat` *Class*

Click here to view code image

```
# File: ChoiceBundle.properties
...
pattern = There {0}.
none = are no bananas
singular = is only one banana
plural = are {1,number,integer} bananas
```

Click here to view code image

```
import java.text.*;
import java.util.*;

public class ChoiceFormatUsage {

  static void displayMessages(Locale requestedLocale)  {
    System.out.println("Requested locale: " + requestedLocale);

    // Fetch the resource bundle:
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // Create formatter for specified pattern and locale:
    MessageFormat mf = new MessageFormat(
        bundle.getString("pattern"),       // pattern = There {0}.
        requestedLocale
        );

    // Create the limits and the formats arrays:
    double[] limits = {0,1,2};                                                 //
    String [] grammarFormats = {                                               //
        bundle.getString("none"),      // none = are no bananas
        bundle.getString("singular"),  // singular = is only one banana
        bundle.getString("plural")     // plural = are {1,number,integer} ban
    };

    // Create the choice format:
    ChoiceFormat choiceForm = new ChoiceFormat(limits, grammarFormats);
    // Create the formats:
    Format[] formats = {choiceForm};

    // Set the formats in the formatter:
    mf.setFormats(formats);                                                //
//  messageForm.setFormatsByArgumentIndex(formats);                        //
//  mf.setFormat(0, choiceForm);                                           //

    // Create the arguments arrays:
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);           //
      System.out.printf("Arguments:%-10sResult: %s%n",
          Arrays.toString(messageArguments[choiceNumber]),result);     }
  }

  public static void main(String[] args)   {
    displayMessages(new Locale("en", "US"));
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

Output from the program:

Click here to view code image

```
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

Parsing Values Using Patterns

An instance of the `MessageFormat` class can be used both for formatting and for parsing of values. The class provides the parse() methods for parsing text. We will primarily use the one-argument method shown below to demonstrate parsing with the `MessageFormat` class.

> Click here to view code image
>
> ```
> Object[] parse(String source) throws ParseException
> Object[] parse(String source, ParsePosition position)
> ```
>
> The first method parses text from the beginning of the string `source` to return an `Object` array with the parsed values.
>
> The second method parses from the `position` in the string

> `source` where the parsing should start.
>
> Both methods may not use the entire text of the given string `source`.

The `parse()` methods return an `Object` array with the parsed values, which means that the parsed values must be extracted from the array and cast to the appropriate type in order to compute with them. The cast to the appropriate type must be done safely—typically by determining the type with the `instanceof` operator before applying the cast.

As the one-argument `parse()` method throws a checked `ParseException` when an error occurs under parsing, the code snippets in this subsection must be executed in a context that handles this exception.

The round trip of formatting values and then parsing the formatted result to obtain the same values back is illustrated below. It should not come as a surprise that this round trip always works, as demonstrated by the code below. We see that the argument of type `double` that was formatted and the value of type `double` that was parsed according to the pattern at (1) are equal.

Click here to view code image

```
String pattern = "foo {0,number,currency} bar";                      // (1)
MessageFormat mfp = new MessageFormat(pattern, Locale.US);
Object[] arguments = new Object[] {10.99};            // [10.99]
String formattedResult = mfp.format(arguments);       // "foo $10.99 bar"
Object[] parsedResult = mfp.parse(formattedResult);   // [10.99]
System.out.println((double)arguments[0] == (double)parsedResult[0]); // true
```

A vanilla format element, `{i}`, in a pattern matches a string in the source, as no format type or style is specified for such a format element. The source at (3) matches the format element in the pattern at (2), as is evident from the parse result at (4).

Click here to view code image

```
String patternA = "{0}";                                             // (2)
MessageFormat mfpA = new MessageFormat(patternA, Locale.US);
Object[] parsedResultA = mfpA.parse("One Two Three");                // (3)
System.out.println(parsedResultA[0] instanceof String);   // true
System.out.println(Arrays.toString(parsedResultA));       // (4) [One Two Thre
```

Any text in a pattern must be matched verbatim in the source. Note how the text in the pattern at (5) is matched in the source at (6), with the string `"2"` being returned as the result of parsing.

Click here to view code image

```
String patternB = "One {0} Three";                                  // (5)
MessageFormat mfpB = new MessageFormat(patternB, Locale.US);
Object[] parsedResultB = mfpB.parse("One 2 Three");                 // (6)
System.out.println(parsedResultB[0].equals("2"));                   // true
System.out.println(Arrays.toString(parsedResultB));                 // [2]
```

The code below illustrates that the `parse()` method may not use the entire text in the source, only what is necessary to declare a match. Parsing stops after the word `Three` in the source at (7) when a match for the pattern has been found in the source.

Click here to view code image

```
Object[] parsedResultC = mfpB.parse("One 2 Three whatever");        // (7)
System.out.println(Arrays.toString(parsedResultC));                 // [2]
```

In order for the parse to succeed, the source must contain text that is compatible with the pattern so that it can be parsed according to any type or style specified in the format elements defined in the pattern. At (8) below, the pattern contains two format elements that require an integer and a currency value specified according to the US locale. The source at (9) meets the requirements to match the pattern. The parse result shows that an `int` value and a `double` value were parsed.

Click here to view code image

```
String patternD = "foo {0,number,integer} {1,number,currency} bar";  // (8)
MessageFormat mfpD = new MessageFormat(patternD, Locale.US);
Object[] parsedResultD = mfpD.parse("foo 2021 $64.99 bar");         // (9)
System.out.println(Arrays.toString(parsedResultD));         // [2021, 64.99]
```

We have seen earlier in this chapter that a *nbsp* is necessary in the source in order to parse certain currencies (p. 1119). That is also the case when parsing such currency values with the `MessageFormat` class, and is illustrated below for the Norwegian locale. Unless a *nbsp* is the delimiter between the currency symbol and the first digit of the amount, as shown at (10b), parsing will result in a checked `Parse-Exception` being thrown.

Click here to view code image

```
Locale locale = new Locale("no", "NO");
String pattern1 = "foo {0,number,currency} bar";
MessageFormat mfp1 = new MessageFormat(pattern1, locale);
// String parseSource = "foo kr 10,99 bar";                // (10a) ParseExcept
String parseSource = "foo kr\u00a010,99 bar";              // (10b) nbsp. OK.
Object[] parsedResults = mfp1.parse(parseSource);
if (parsedResults[0] instanceof Double dValue) {
  System.out.println(dValue);                              // 10.99
}
```

Additional Support for Formatting in the Java SE Platform APIs

In this subsection we mention additional support for formatting values that is provided by various classes in the Java SE Platform APIs.

The class `java.util.Formatter` provides the core support for *formatted text representation* of primitive values and objects through its overloaded `format()` methods. See the Java SE Platform API documentation for the nitty-gritty details on how to specify the format.

> Click here to view code image
>
> ```
> format(String format, Object... args)
> format(Locale loc, String format, Object... args)
> ```
>
> Write a string that is a result of applying the specified `format` string to the values in the variable arity parameter `args`. The resulting string is written to the *destination object* that is associated with the formatter.
>
> The destination object of a formatter is specified when the formatter is created. The destination object can, for example, be a `String`, a `StringBuilder`, a file, or any `OutputStream`.
>
> Return the current formatter.

The classes `java.io.PrintStream` and `java.io.PrintWriter` also provide an overloaded `format()` method with the same signature for formatted output. These streams use an associated `Formatter` that sends the output to the `PrintStream` or the `PrintWriter`, respectively. We have used the `printf()` method of the `PrintStream` class for sending output to the terminal every time this method is invoked on the `System.out` field (§1.9, p. 24).

The `String` class also provides an analogous `format()` method, but it is static. This method also uses an associated `Formatter` for formatting purposes. Unlike the `format()` method of the classes mentioned earlier, this static method returns the resulting string after formatting the values (§8.4, p. 457).

The `java.io.Console` only provides the first form of the `format()` and the `printf()` methods (without the locale specification). These methods too use an associated `Formatter`. They write the resulting string to the console's output stream, and return the current console (§20.4, p. 1256).

Review Questions

18.5 Given the following code:

Click here to view code image

```
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Locale;

public class DecimalNumberPatternsRQ {
  public static void main(String[] args) {
    // (1) INSERT CODE HERE
    double value = 0.456;
    DecimalFormat df = (DecimalFormat) NumberFormat.getNumberInstance(Lo
    df.applyPattern(pattern);
    String output = df.format(value);
    System.out.printf("|%s|", output);
  }
}
```

Which code, when inserted independently at (1), will result in the following output: `|.46|`?

Select the five correct answers.

a. `String pattern = ".00";`

b. `String pattern = ".##";`

c. `String pattern = ".0#";`

d. `String pattern = "#.00";`

e. `String pattern = "#.0#";`

f. `String pattern = "#.##";`

g. `String pattern = ".#0";`

18.6 Given the following code:

Click here to view code image

```
import java.math.*;
import java.text.*;
import java.util.*;
public class RQ7 {
  public static void main(String[] args) {
    NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);
    nf.setMaximumFractionDigits(2);
    nf.setRoundingMode(RoundingMode.HALF_DOWN);
    double value = 9876.54321;
    String s1 = nf.format(value);
    nf.setRoundingMode(RoundingMode.HALF_UP);
    String s2 = nf.format(value);
    System.out.println(s1 + " " + s2);
  }
}
```

What is the result?

Select the one correct answer.

a. `$9,876.54 $9,876.55`

b. `$9,876.53 $9,876.53`

c. `$9,876.54 $9,876.54`

d. `$9,876.53 $9,876.55`

e. The program will throw an exception at runtime.

18.7 Given the following code:

Click here to view code image

```
import java.math.*;
import java.text.*;
import java.util.*;
public class RQ7Alt {
  public static void main(String[] args) {
    BigDecimal value = new BigDecimal("9876.545");
    NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);
    nf.setMaximumFractionDigits(2);
    nf.setRoundingMode(RoundingMode.HALF_DOWN);
    String s1 = nf.format(value);
    System.out.println();
    nf.setRoundingMode(RoundingMode.HALF_UP);
    String s2 = nf.format(value);
    System.out.println(s1+" "+s2);
  }
}
```

What is the result?

Select the one correct answer.

a. `$9,876.55 $9,876.55`

b. `$9,876.54 $9,876.54`

c. `$9,876.55 $9,876.54`

d. `$9,876.54 $9,876.55`

e. The program will throw an exception at runtime.

18.8 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.*;
public class RQ8 {
  public static void main(String[] args) {
    LocalDate foolsDay = LocalDate.of(2021, Month.APRIL, 1);
    DateTimeFormatter df = DateTimeFormatter
                             .ofLocalizedDateTime(FormatStyle.SHORT)
                             .localizedBy(Locale.UK);
    System.out.print(df.format(foolsDay));
  }
}
```

What is the result?

Select the one correct answer.

a. `04/01/2021`

b. `01/04/2021`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

18.9 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.*;
public class RQ9 {
  public static void main(String[] args) {
    LocalDateTime foolsDay = LocalDateTime.of(2021, Month.APRIL, 1, 14,
    DateTimeFormatter df = DateTimeFormatter.ofPattern("day")
                                            .localizedBy(Locale.UK);
    System.out.print(df.format(foolsDay));
  }
}
```

What is the result?

Select the one correct answer.

a. `1`
b. `01`
c. `1am`
d. `1am21`
e. `1am2021`
f. `1pm`
g. `1pm21`
h. `1pm2021`
i. The program will throw an exception at runtime.

18.10 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.*;
public class RQ10 {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(2021, Month.APRIL, 1);
    Locale.setDefault(Locale.UK);
    DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle
    String s1 = df.format(d);
    Locale.setDefault(Locale.FRANCE);
    String s2 = df.format(d);
    df.localizedBy(Locale.US);
    String s3 = df.format(d);
    if("Apr 1, 2021".equals(s1)) {
      System.out.print("UK ");
    }
    if("1 avr. 2021".equals(s2)) {
      System.out.print("FR ");
    }
    if("1 Apr 2021".equals(s3)) {
      System.out.println("US");
    }
  }
}
```

What is the result?

Select the one correct answer.

a. `UK FR US`

b. `UK US`

c. `UK FR`

d. `FR US`

e. `UK`

f. `FR`

g. `US`

18.11 Given the following code:

Click here to view code image

```
import java.text.*;
import java.util.*;
public class RQ11 {
  public static void main(String[] args) {
    double x = 0.987654321;
    NumberFormat nf = NumberFormat.getPercentInstance(Locale.US);
    System.out.println(nf.format(x));
  }
}
```

What is the result?

Select the one correct answer.

a. `0.98%`

b. `0.99%`

c. `98%`

d. `99%`

e. `0.987654321%`

f. `987654321%`

18.12 Given the following code:

Click here to view code image

```
import java.text.*;
import java.time.*;
import java.time.format.*;
public class RQ12 {
  public static void main(String[] args) {
    ZoneId london = ZoneId.of("Europe/London");
    ZoneId paris = ZoneId.of("Europe/Paris");
    LocalDateTime date1 = LocalDateTime.parse("2021-01-01T01:01:01",
                                              DateTimeFormatter.ISO_DATE
    ZonedDateTime date2 = date1.atZone(london);
    DateTimeFormatter df1 = DateTimeFormatter.ofPattern("hz")
                                             .withZone(london);
    DateTimeFormatter df2 = DateTimeFormatter.ofPattern("hz")
                                             .withZone(paris);
    String result = MessageFormat.format("{0} {1} {2} {3}",
                                         df1.format(date1), df1.format(
                                         df2.format(date1), df2.format(
    System.out.println(result);
  }
}
```

Given that London (GMT) and Paris (CET) time zones are exactly one hour apart, what is the result?

Select the one correct answer.

a. `1GMT 1GMT 1CET 2CET`

b. `1GMT 2GMT 1CET 2CET`

c. `1GMT 1CET 1CET 2CET`

d. `1GMT 2CET 1GMT 2CET`

e. `1GMT 1GMT 2CET 2CET`

18.13 Given the following code:

Click here to view code image

```
import java.text.MessageFormat;
public class RQ13 {
  public static void main(String[] args) {
    String a = "A", b = "B";
    String result = MessageFormat.format("{0}-'{1}'-{3}-{0}-{1}-'{2}'",
    System.out.println(result);
  }
}
```

What is the result?

Select the one correct answer.

a. `A-{1}--A-B-{2}`

b. `A-B--A-B-{2}`

c. `A-B-{3}-A-B-`

d. `A-{1}-{3}-A-B-{2}`

e. The program will throw an exception at runtime.

18.14 Given the following code:

Click here to view code image

```
import java.text.*;
public class RQ14 {
  public static void main(String[] args) {
    double[] limits = {0,1,2,3,4};
    String[] formats = {"zero","{1}st","{1}nd","{1}rd","{1}th"};
    ChoiceFormat cf = new ChoiceFormat(limits, formats);
    MessageFormat mf = new MessageFormat("{0}");
    mf.setFormat(0, cf);
    Object[] values = {4,5};
    System.out.println(mf.format(values));
  }
}
```

What is the result?

Select the one correct answer.

a. `4th`

b. `5th`

c. `{1}th`

d. The program will throw an exception at runtime.

18.15 Given the following code:

Click here to view code image

```
import java.text.*;
public class RQ15 {
  public static void main(String[] args) {
    double[] limits = {0,-1,1};
    String[] formats = {"zero","negative","positive"};
    ChoiceFormat cf = new ChoiceFormat(limits, formats);
    MessageFormat mf = new MessageFormat("{0}");
    mf.setFormat(0, cf);
    Object[] values = {0.9};
    System.out.println(mf.format(values));
  }
}
```

What is the result?

Select the one correct answer.

a. `zero`

b. `negative`

c. `positive`

d. The program will throw an exception at runtime.

18.16 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.*;
public class RQ16 {
  public static void main(String[] args) {
    Locale.setDefault(Locale.US);
    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyl
    LocalDate d = LocalDate.parse("Apr 1, 2021", dtf);   // (1)
    d = LocalDate.parse("2021-04-01");                   // (2)
    String s = d.format(dft);                            // (3)
    System.out.println(s);
  }
}
```

What is the result?

Select the one correct answer.

a. `Apr 1, 2021`

b. `2021-04-01`

c. The program will throw a runtime exception at (1).

d. The program will throw a runtime exception at (2).

e. The program will throw a runtime exception at (3).

18.17 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.*;
import java.util.*;
public class RQ17 {
  public static void main(String[] args) {
    Locale.setDefault(Locale.US);
    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyl
    LocalDate d = LocalDate.parse("2021-04-01");     // (1)
```

```
        d = LocalDate.parse("4/1/21", dtf);                // (2)
        System.out.println(d);
      }
    }
```

What is the result?

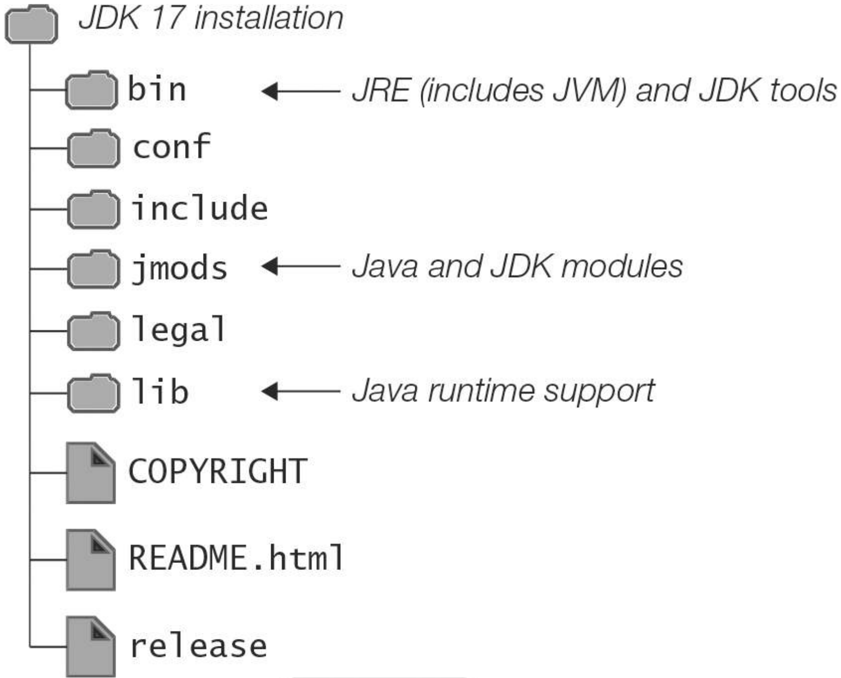
Select the one correct answer.

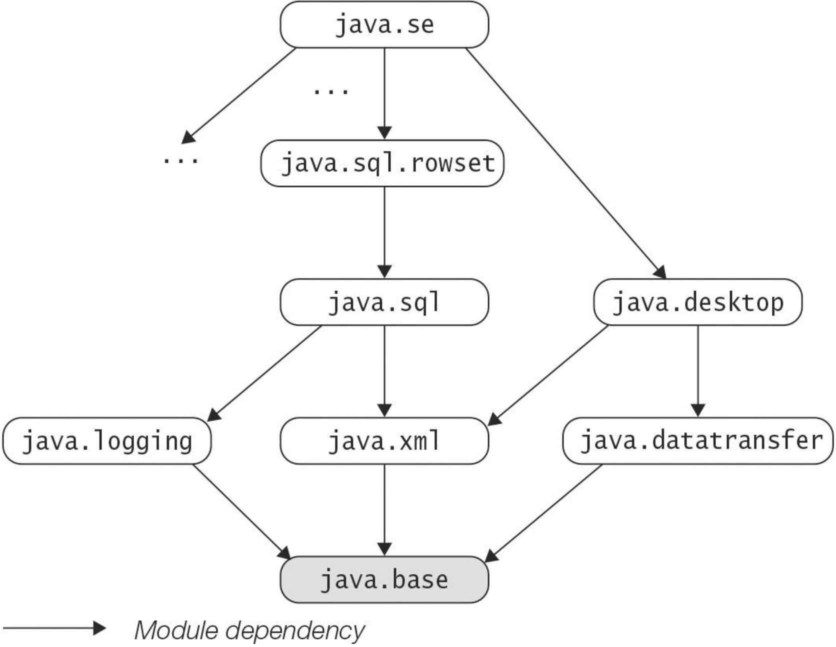
a. `Apr 1, 2021`
b. `4/1/21`
c. `2021-04-01`
d. The program will throw a runtime exception at (1).
e. The program will throw a runtime exception at (2).

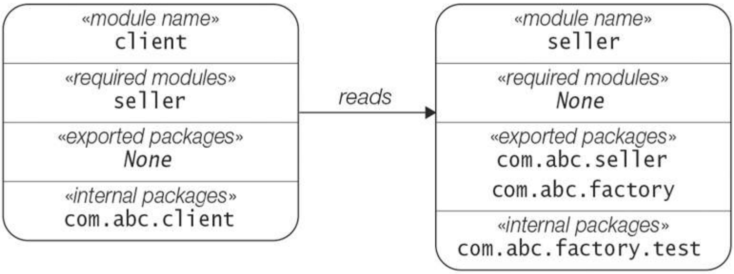
18.18 Which statement is true about localization?

Select the one correct answer.

a. The default locale is `Locale.US`.
b. The default locale is `Locale.ISO`.
c. The default format for the `LocalDate` object is `DateTimeFormatter.BASIC_ISO_DATE`.
d. The default format for the `LocalDate` object is `DateTimeFormatter.ISO_DATE`.


JDK 17 installation
bin
JRE (includes JVM) and JDK tools
conf
include
jmods
Java and JDK modules
legal
lib
Java runtime support
COPYRIGHT
README.html
release


java.se
...
...
java.sql.rowset
java.sql
java.desktop
java.logging
java.xml
java.datatransfer
java.base
Module dependency


«module name»
client
«required modules»
seller
«exported packages»
None
«internal packages»
com.abc.client
reads
«module name»
seller
«required modules»
None
«exported packages»
com.abc.seller
com.abc.factory
«internal packages»
com.abc.factory.test

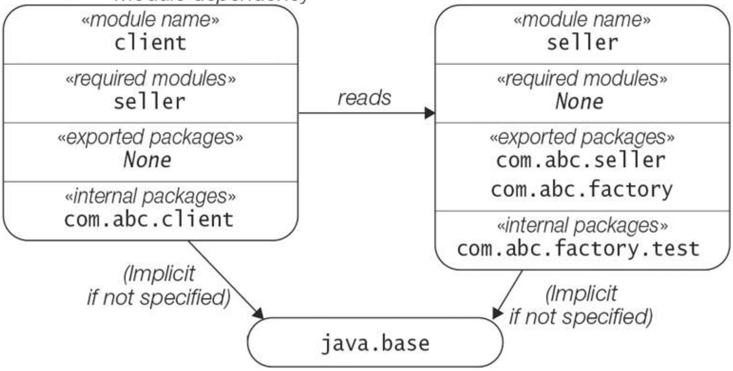
(a) Module `client` (b) Module `seller`

```
module client {
  requires seller;
}
```

```
module seller {
  exports com.abc.seller;
  exports com.abc.factory;
}
```

(c) Module declaration for module `client` (d) Module declaration for module `seller`

→ Module dependency


«module name»
client
«required modules»
seller
«exported packages»
None
«internal packages»
com.abc.client
reads
«module name»
seller
«required modules»
None
«exported packages»
com.abc.seller
com.abc.factory
«internal packages»
com.abc.factory.test
(Implicit
if not specified)
(Implicit
if not specified)
java.base

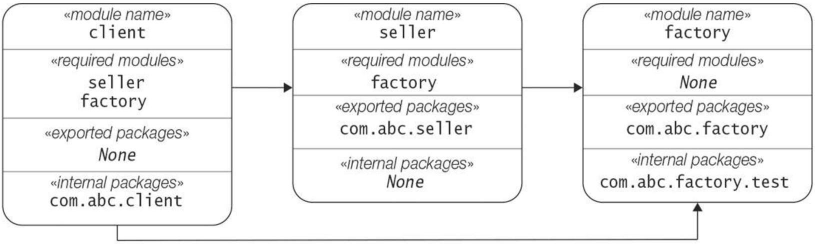
(a) Module graph

```
module client {
  requires seller;
  requires java.base;
}
```

```
module seller {
  requires java.base;
  exports com.abc.seller;
  exports com.abc.factory;
}
```

(b) Module declaration for module `client` (c) Module declaration for module `seller`

→ Module dependency


«module name»
client
«required modules»
seller
factory
«exported packages»
None
«internal packages»
com.abc.client
«module name»
seller
«required modules»
factory
«exported packages»
com.abc.seller
«internal packages»
None
«module name»
factory
«required modules»
None
«exported packages»
com.abc.factory
«internal packages»
com.abc.factory.test

(a) Module `client` *(b) Module* `seller` *(c) Module* `factory`

```
module client {
  requires seller;
  requires factory;
}
```

(d) Module declaration for `client`

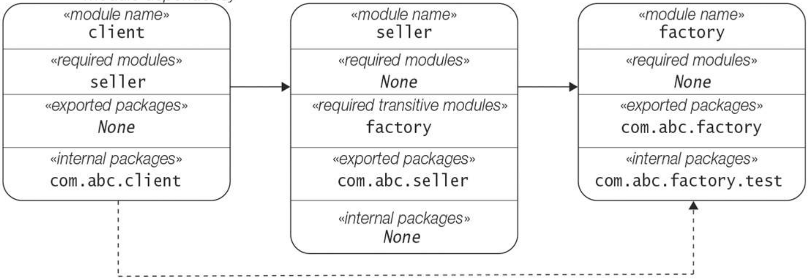
```
module seller {
  requires factory;
  exports com.abc.seller;
}
```

(e) Module declaration for `seller`

```
module factory {
  exports com.abc.factory;
}
```

(f) Module declaration for `factory`

→ *Module dependency*


«module name»
client
«required modules»
seller
«exported packages»
None
«internal packages»
com.abc.client
«module name»
seller
«required modules»
None
«required transitive modules»
factory
«exported packages»
com.abc.seller
«internal packages»
None
«module name»
factory
«required modules»
None
«exported packages»
com.abc.factory
«internal packages»
com.abc.factory.test

(a) Module `client` *(b) Module* `seller` *(c) Module* `factory`

```
module client {
  requires seller;
}
```

(d) Module declaration for `client`

```
module seller {
  requires transitive factory;
  exports com.abc.seller;
}
```

(e) Module declaration for `seller`

```
module factory {
  exports com.abc.factory;
}
```

(f) Module declaration for `factory`

→ *Module dependency* ⇢ *Implied module dependency*

```
module client {
  requires seller;
  requires factory;
}
```

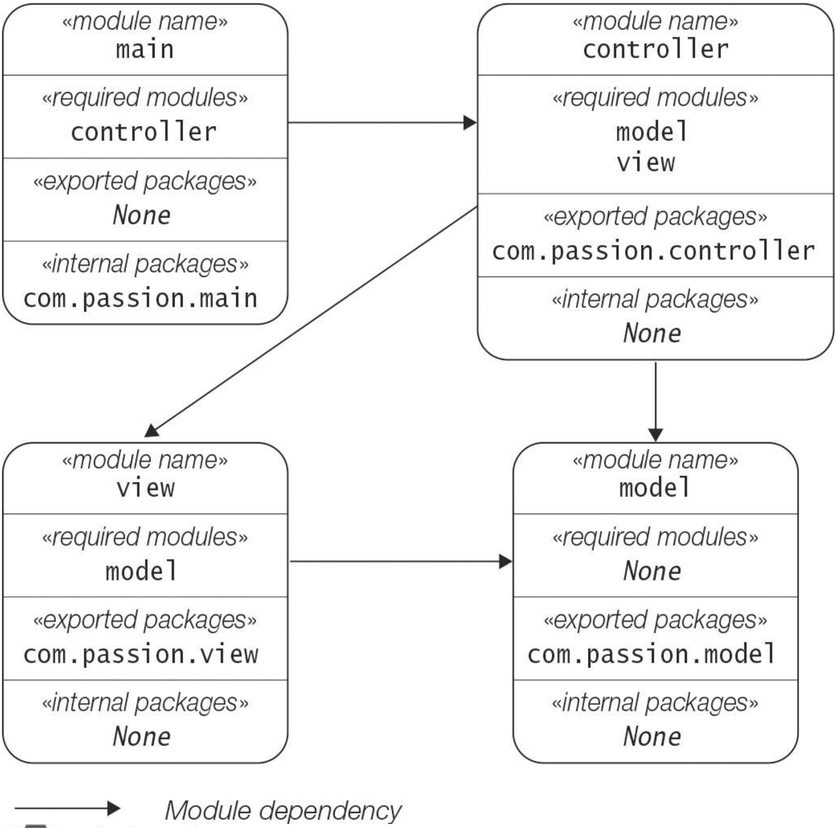
(d') Module declaration for `client`

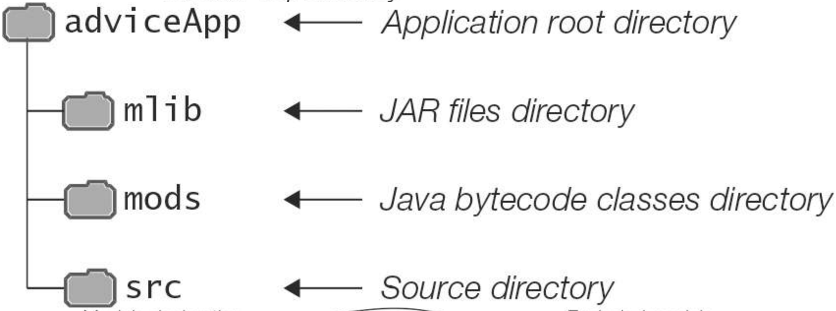
```
module seller {
  requires factory;
  exports com.abc.seller;
}
```

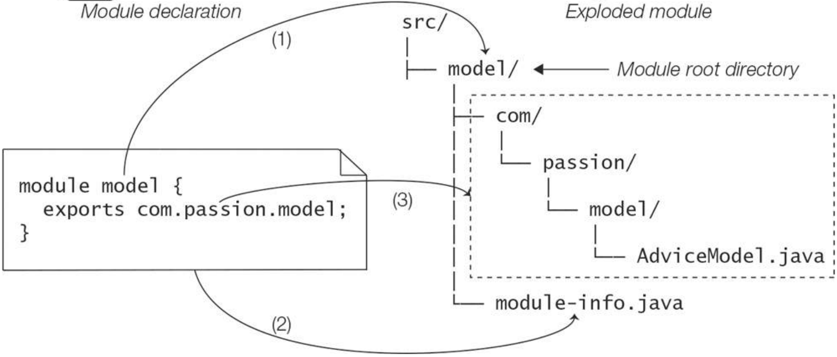
(e') Module declaration for `seller`

```
module factory {
  exports com.abc.factory
      to seller, client;
}
```

(f') Module declaration for `factory` *with qualified export of its package*

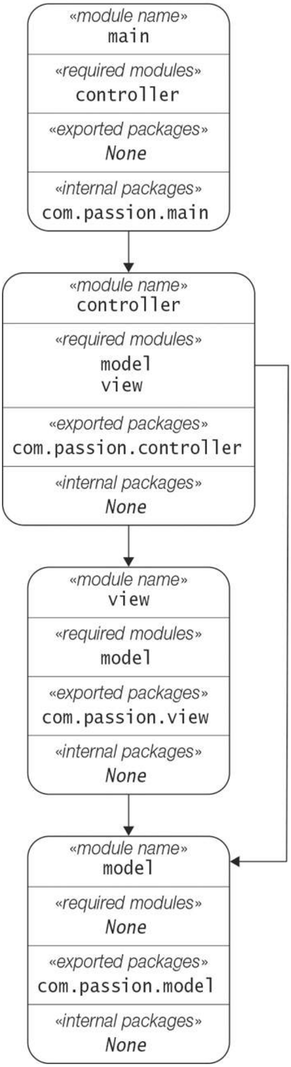

«module name»
main
«required modules»
controller
«exported packages»
None
«internal packages»
com.passion.main
«module name»
controller
«required modules»
model
view
«exported packages»
com.passion.controller
«internal packages»
None
«module name»
view
«required modules»
model
«exported packages»
com.passion.view
«internal packages»
None
«module name»
model
«required modules»
None
«exported packages»
com.passion.model
«internal packages»
None
Module dependency


adviceApp
Application root directory
mlib
JAR files directory
mods
Java bytecode classes directory
src
Source directory


Module declaration
Exploded module
(1)
(2)
(3)
src/
model/
Module root directory
com/
passion/
model/
AdviceModel.java
module-info.java

```
module model {
  exports com.passion.model;
}
```

Module graph


«module name»
main
«required modules»
controller
«exported packages»
None
«internal packages»
com.passion.main
«module name»
controller
«required modules»
model
view
«exported packages»
com.passion.controller
«internal packages»
None
«module name»
view
«required modules»
model
«exported packages»
com.passion.view
«internal packages»
None
«module name»
model
«required modules»
None
«exported packages»
com.passion.model
«internal packages»
None

Module declarations

```
// File path: src/main/module-info.java
module main {
  requires controller;
}
```

(a)

```
// File path: src/controller/module-info.java
module controller {
  requires model;
  requires view;
  exports com.passion.controller;
}
```

(b)

```
// File path: src/view/module-info.java
module view {
  requires model;
  exports com.passion.view;
}
```

(c)

```
// File path: src/model/module-info.java
module model {
  exports com.passion.model;
}
```

(d)

```
src/
├── controller/
│   ├── com/
│   │   └── passion/
│   │       └── controller/
│   │           └── AdviceController.java
│   └── module-info.java
├── main/
│   ├── com/
│   │   └── passion/
│   │       └── main/
│   │           └── Main.java
│   └── module-info.java
├── model/
│   ├── com/
│   │   └── passion/
│   │       └── model/
│   │           └── AdviceModel.java
│   └── module-info.java
└── view/
    ├── com/
    │   └── passion/
    │       └── view/
    │           └── AdviceView.java
    └── module-info.java
```

(a) Exploded modules containing source code

```
(3)
>javac --module-path mods \
      -d mods/controller \
      src/controller/module-info.java \
      src/controller/com/passion/ \
        controller/AdviceController.java
(4)
>javac --module-path mods \
      -d mods/main \
      src/main/module-info.java \
      src/main/com/passion/main/ \
                      Main.java
(1)
>javac --module-path mods \
      -d mods/model \
      src/model/module-info.java \
      src/model/com/passion/model/ \
                  AdviceModel.java
(2)
>javac --module-path mods \
      -d mods/view \
      src/view/module-info.java \
      src/view/com/passion/view/ \
                 AdviceView.java
```

(b) Order of compiling exploded modules individually

```
src/
└── model/
    ├── com/
    │   └── passion/
    │       └── model/
    │           └── AdviceModel.java
    └── module-info.java
```

(a) Exploded module with source files

```
mods/
└── model/
    ├── com/
    │   └── passion/
    │       └── model/
    │           └── AdviceModel.class
    └── module-info.class
```

(b) Exploded module with Java bytecode classes

```
>javac --module-path mods \
      -d mods/model \
      src/model/module-info.java \
      src/model/com/passion/model/AdviceModel.java
```

(c) Command to compile the `model` module

```
mods/
├── controller/
│   ├── com/
│   │   └── passion/
│   │       └── controller/
│   │           └── AdviceController.class
│   └── module-info.class
├── main/
│   ├── com/
│   │   └── passion/
│   │       └── main/
│   │           └── Main.class
│   └── module-info.class
├── model/
│   ├── com/
│   │   └── passion/
│   │       └── model/
│   │           └── AdviceModel.class
│   └── module-info.class
└── view/
    ├── com/
    │   └── passion/
    │       └── view/
    │           └── AdviceView.class
    └── module-info.class
```

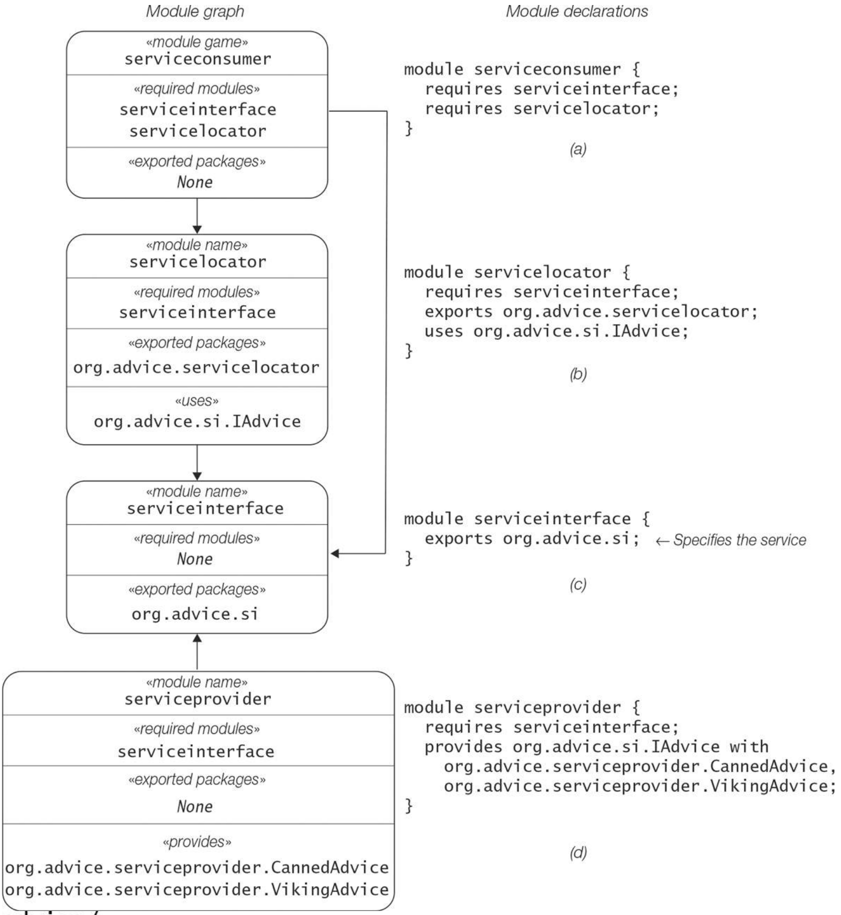
(a) Exploded modules with Java bytecode classes

```
>jar --verbose \
     --create \
     --file mlib/controller.jar \
      -C mods/controller .

>jar --verbose \
     --create \
     --file mlib/main.jar \
     --main-class com.passion.main.Main \
      -C mods/main .

>jar --verbose \
     --create \
     --file mlib/model.jar \
      -C mods/model .

>jar --verbose \
     --create \
     --file mlib/view.jar \
      -C mods/view .
```

(b) Creating JAR files

```
mlib/
├── controller.jar
├── main.jar
├── model.jar
└── view.jar
```

(c) JAR file directory


Module graph
Module declarations
«module game»
serviceconsumer
«required modules»
serviceinterface
servicelocator
«exported packages»
None
«module name»
servicelocator
«required modules»
serviceinterface
«exported packages»
org.advice.servicelocator
«uses»
org.advice.si.IAdvice
«module name»
serviceinterface
«required modules»
None
«exported packages»
org.advice.si
«module name»
serviceprovider
«required modules»
serviceinterface
«exported packages»
None
«provides»
org.advice.serviceprovider.CannedAdvice
org.advice.serviceprovider.VikingAdvice
← Specifies the service

```
module serviceconsumer {
  requires serviceinterface;
  requires servicelocator;
}
```

(a)

```
module servicelocator {
  requires serviceinterface;
  exports org.advice.servicelocator;
  uses org.advice.si.IAdvice;
}
```

(b)

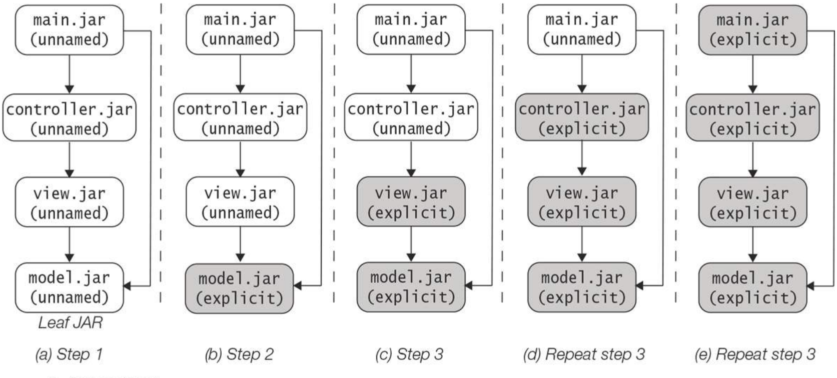
```
module serviceinterface {
  exports org.advice.si;
}
```

(c)

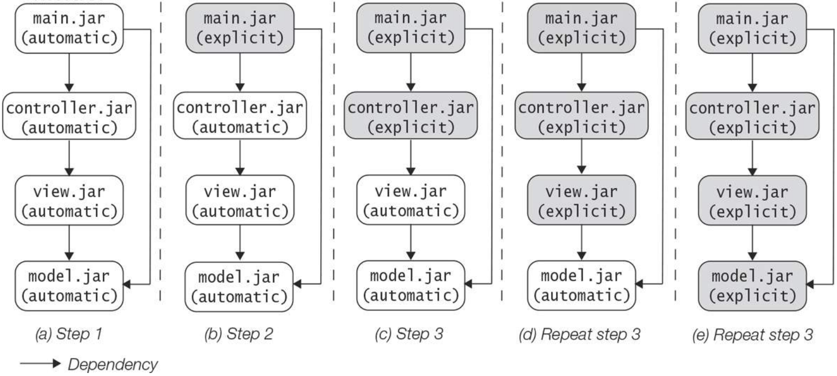
```
module serviceprovider {
  requires serviceinterface;
  provides org.advice.si.IAdvice with
    org.advice.serviceprovider.CannedAdvice,
    org.advice.serviceprovider.VikingAdvice;
}
```

(d)

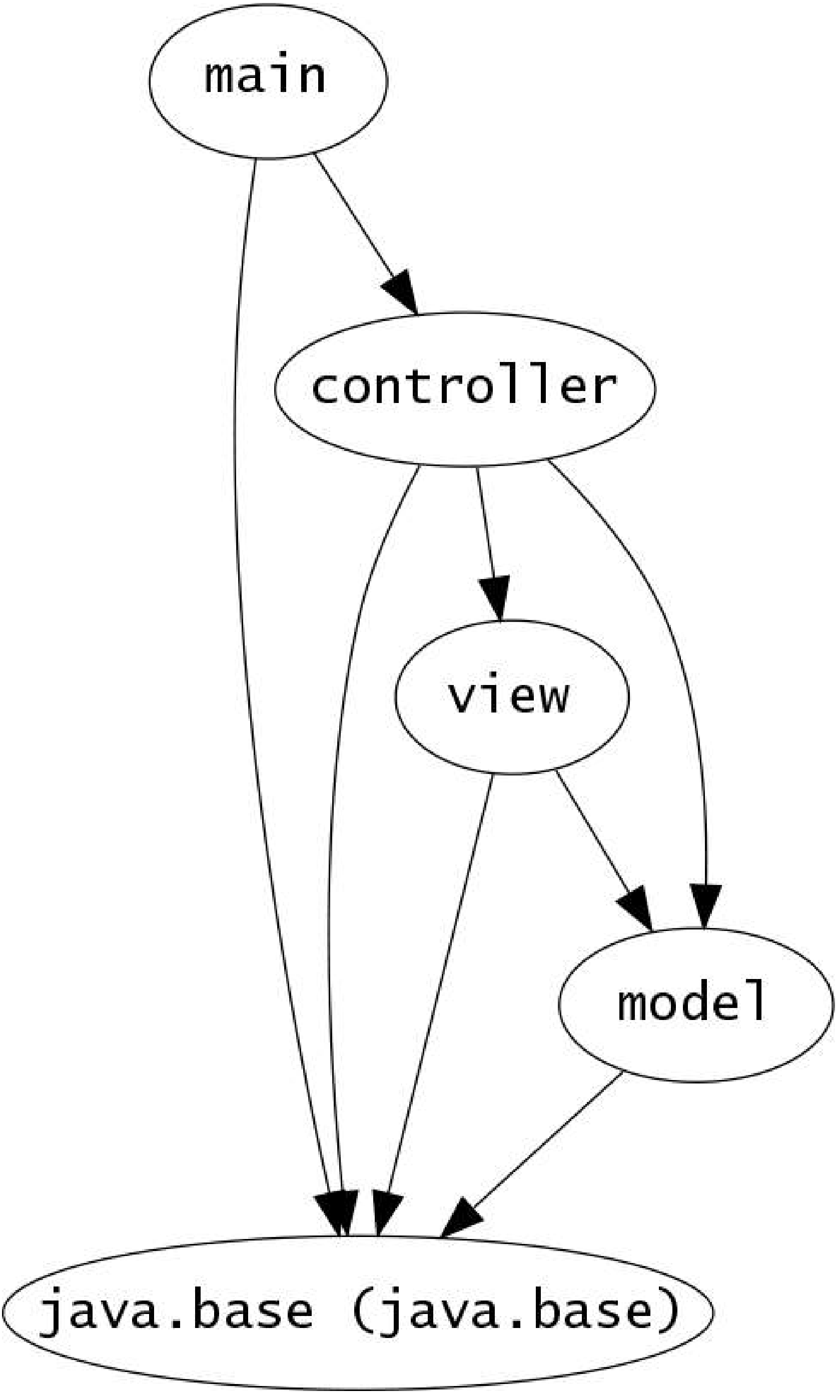
```
advice/
├── bin/
|   ├── java            <=== Java command to launch the application
|   └── ...
|
├── conf/
|
├── include/
|
├── legal/
|
├── lib/
|   ├── jrt-fs.jar       <=== Java runtime file system
|   ├── modules          <=== Container for application modules
|   └── ...
|
└── release
```


main.jar
(unnamed)
controller.jar
(unnamed)
view.jar
(unnamed)
model.jar
(unnamed)
Leaf JAR
main.jar
(unnamed)
controller.jar
(unnamed)
view.jar
(unnamed)
model.jar
(explicit)
main.jar
(unnamed)
controller.jar
(unnamed)
view.jar
(explicit)
model.jar
(explicit)
main.jar
(unnamed)
controller.jar
(explicit)
view.jar
(explicit)
model.jar
(explicit)
main.jar
(explicit)
controller.jar
(explicit)
view.jar
(explicit)
model.jar
(explicit)
Dependency

(a) Step 1 (b) Step 2 (c) Step 3 (d) Repeat step 3 (e) Repeat step 3


Root JAR
main.jar
(automatic)
controller.jar
(automatic)
view.jar
(automatic)
model.jar
(automatic)
main.jar
(explicit)
controller.jar
(automatic)
view.jar
(automatic)
model.jar
(automatic)
main.jar
(explicit)
controller.jar
(explicit)
view.jar
(automatic)
model.jar
(automatic)
main.jar
(explicit)
controller.jar
(explicit)
view.jar
(explicit)
model.jar
(automatic)
main.jar
(explicit)
controller.jar
(explicit)
view.jar
(explicit)
model.jar
(explicit)
Dependency

(a) Step 1 (b) Step 2 (c) Step 3 (d) Repeat step 3 (e) Repeat step 3


main
controller
view
model
java.base (java.base)

Java Module System 19

Chapter Topics

- Understanding the issues the modules address and the benefits they provide
- Understanding the most important goals of the Java Module System: strong encapsulation, reliable configuration, and better performance
- Understanding how the modular JDK is organized, and how to discover and use its salient modules
- Writing a module declaration that specifies the properties of a module: its module name, and the `requires` and `exports` directives in such a declaration
- Understanding the module graph of an application
- Differentiating between readability of a module and accessibility of `public` types in an exported package of a required module
- Understanding the accessibility rules for members of a `public` type whose package is exported
- Understanding how implied dependencies are introduced in the module graph by the `requires transitive` directive, and its implication
- Creating the directory structure of a modular application, where an exploded module contains the source code for all packages in a module
- Compiling and running a multi-module application
- Creating modular JARs as repositories of compiled modules

- Running an application from its modular JARs
- Implications of open modules and the `opens` directives for reflection at runtime
- Creating, loading, and using services with `provides-with` and `uses` directives: specifying a service interface, and implementing service provider, service locator, and service consumer
- Creating and executing runtime images
- Differentiating between different kinds of modules: unnamed, automatic, and named modules
- Applying code migration strategies to modularize plain code
- Exploring modular JARs: listing JAR contents, extracting JAR contents, listing observable modules, describing the module descriptor, and viewing module-level, package-level, and class-level dependencies
- Providing a summary of relevant options accepted by the JDK command-line tools (`javac`, `java`, `jar`, `jdeps`, and `jlink`)

Java SE 17 Developer Exam O	
[7.1] Define modules and their dependencies, expose module content including for reflection. Define services, producers, and consumers	*§19.1, p. 1163* *to* *§19.5, p. 1179* *§19.8, p. 1191* *§19.9, p. 1196*
[7.2] Compile Java code, produce modular and non-modular jars, runtime images, and implement migration using unnamed and automatic modules	*§19.6, p. 1186* *§19.7, p. 1189* *§19.10, p. 1204* *to* *§19.14, p. 1218*
Java SE 11 Developer Exam O	
[7.1] Deploy and execute modular applications, including automatic modules	*§19.2, p. 1164* *§19.6, p. 1186* *§19.7, p. 1189* *§19.10, p. 1204* *to* *§19.13, p. 1214*
[7.2] Declare, use, and expose modules, including the use of services	*§19.3, p. 1168* *§19.4, p. 1177* *§19.5, p. 1179*

§19.8, p. 1191 *§19.9, p. 1196*	

The *Java Platform Module System* (JPMS) provides the software engineering technology that makes it possible to efficiently build, deploy, maintain, and evolve very large software systems. It was successfully used to modularize the JDK before finally being released in Java 9.

This chapter discusses how the language, the JDK tools, and the runtime environment provide the support for creating modular applications in Java.

19.1 Making the Case for Modules

We begin by briefly examining the issues that modules address and the benefits they provide.

Stronger Encapsulation

Modules in Java provide a new level of encapsulation above packages. Whereas a package encapsulates classes and interfaces, a module encapsulates a set of packages and any other resources required by the module. An application is thus composed of reusable and related modules, where each module groups a set of packages and each package groups a set of classes and interfaces.

Fine-Grained Visibility of Public Types

The need for modules in Java arose from many concerns. For programming in the large, the concept of packages has many shortcomings, particularly in providing more fine-grained visibility of public types which are accessible in *all* packages, with no possibility of curtailing the accessibility of such a public type. A module can provide stronger encapsulation with a well-defined public API that other modules can use, and packages that are meant to be internal to a module are now guaranteed to be inaccessible to other modules.

Reusability, Maintainability, and Optimization

Applying the principles of encapsulation at each encapsulation level increases the reusability and aids the maintainability of the code, resulting in modules that can be optimized independently.

Reliable Configuration

Prior to modules, the Java language did not support any notion of *dependencies* between artifacts that comprise an application. Ad hoc use of JARs (Java Archives) proved inadequate for such relationships, as the compiler and the JVM saw the JAR files merely as containers of types and resources.

Now there is better encapsulation of modules in JARs, with one module in each JAR. The Java Module System explicitly utilizes the dependencies between modules to reason and maintain these dependencies both at compile time and at runtime, resulting in reliable application configurations. Before an application is launched, all modules needed by the application are resolved in order to avoid problems relating to missing modules or types during execution.

Better Performance

Modules in Java contribute to developing scalable systems with improved performance, as modules can be optimized independently. The module system guarantees that all classes of the same package are in the same module—that is, there are no *split packages*, and the modules describe explicit dependencies. The class loader thus knows which module to look into when loading a class, improving the startup time of applications.

Scalable Applications

As the packages in the Java API are now organized into modules, it is possible for an application to include only those modules that are needed, thus reducing the binary footprint of the application. The modular JDK now caters to customized Java builds that can be scaled to small devices, and can also be shipped with the runtime environment included.

Improved Security

Organizing an application in modules can also improve security as critical code can be strongly encapsulated in modules, potentially decreasing the attack surface of an application.

Backward Compatibility

Modules are not a mandatory requirement for an application to run in the modular JDK. However, the benefits of using modules for large-scale application development are substantial as mentioned in this section. The modular JDK provides support to incrementally migrate legacy code into modules.

19.2 The Modular JDK

Support for modules is provided by the Java language and the JDK. The concept of a *module* is incorporated into the language. Existing tools in the JDK, such as `javac`, `java`, `jar`, and `jdeps`, have been enhanced to aid module-based development. Support for modules provided by these tools is covered in later sections. This section provides an overview of how the modular JDK is organized and what it has to offer.

Reasons for Modularity

The need to modularize the JDK arose from many reasons, among them the following:

- A monolithic runtime library (epitomized by the `rt.jar` file) was not conducive to the maintainability and evolution of the Java ecosystem with its ever-increasing APIs.
- Bloat of obsolete technologies could not be addressed properly, and was often at the risk of breaking backward compatibility.
- Maintaining backward compatibility in the face of ever-evolving packages made this goal more difficult.
- The one-size-fits-all runtime library started to incur a penalty in terms of increased space, more memory at runtime, and a need for lower consumption of CPU resources.
- JARs as containers of Java code were inadequate to elicit relationships between packages, having no means to expose their dependencies other than by resorting to third-party solutions (e.g., Maven and OSGi).

- The class-path solution to specify the location of package repositories was not optimal in locating types, leading to unforeseen problems when loading them and often throwing an exception at runtime because some type was missing.

Goals of the Modular JDK

Specific goals of the modular JDK, not surprisingly, align with the benefits of modules extolled in §19.1, p. 1163.

- *Stronger encapsulation*: This results in well-defined communication boundaries between modules and stricter encapsulation of their internal implementation.
- *Reliable configuration*: Any combination of modules must satisfy all dependencies both at compile time and runtime.
- *Enhanced performance*: Modularized code is more amiable to program optimizations.
- *Improved scalability*: The evolution of the Java platform can continue in a modularized fashion, as modules can be worked on in parallel.
- *Better security*: Modularized code decreases the attack surface, and access through *reflection* is strictly controlled at runtime.

Overview of the Modular JDK

The monolithic library of the JDK has now been split into many modules, with each module defining a specific functionality of the JDK. The installation directory of the JDK, shown in Figure 19.1, gives an overview of how the JDK is organized. The path of the installation directory will vary depending on the OS platform.

Figure 19.1 *Structure of JDK 17 Installation*

- The `bin` directory: This is where the executables are found—the Java Runtime Environment (JRE), which includes the Java Virtual Machine (JVM), and the JDK command-line tools, among others, that are necessary to develop and run Java programs.
- The `jmods` directory: This directory epitomizes the modular JDK. It contains the *platform modules* of the JDK. These are

compiled modules used to create custom runtime images of applications.

Each platform module is archived in a file having the name of the module and the extension `.jmod`—for example, `java.base.jmod`, `java.se.jmod`, and `jdk.jartool.jmod`. The modules beginning with the prefix `"java."` implement the Java SE specification, often called *standard modules*. These are present in all implementations of the Java SE platform. The modules beginning with the prefix `"jdk."` are platform specific, often called *non-standard modules*. These are not necessarily available for all implementations of the Java SE platform.

JMOD archives use a special archive format reserved for the platform modules, that are created using the JDK tool `jmod`. All JDK tools understand JMOD archives. The platform modules can be explored like any other application module. Examples of exploring such modules can be found in §19.13, p. 1211. There is seldom any need to create a JMOD archive or change the contents of the `jmods` directory.

Note that modules can be packaged both as JARs and as JMOD archives.

- The `lib` directory: This directory contains additional class libraries and support required by the JDK. These files are not meant for external use.

Module Graph of the JDK

The modules in the JDK define *dependencies* on other modules. These dependencies can be viewed as a graph in which the modules are the *nodes* and the *unidirectional edges* between the nodes define the dependencies between the modules. A partial *module graph* of the JDK is shown in Figure 19.2. The complete module graph of the JDK is of course substantially larger, but Figure 19.2 will suffice to convey its modular structure. It is also a *reduced graph* where by redundant dependencies, like the *implicit dependencies* on the `java.base` module (p. 1167), have been omitted.

Figure 19.2 *Partial Module Graph of Java SE 17 Modules*

A unidirectional edge, for example, from the module `java.se` to the `java.desktop` module signifies that the former has a dependency on the latter. The (complete) module graph of the JDK allows us to read which modules any given module depends on. The structure of the module graph in Figure 19.2 highlights two important modules: `java.se` at the zenith and `java.base` at the base of the graph, showing that the former depends on each module in the Java SE platform configuration and the latter on none of them.

Module graphs visualize the modular structure of an application, providing insights into its architecture and in discovering any anomalies. Such graphs can be built using the JDK tool `jdeps` (p. 1214). Other JDK tools, such as `javac` and `java`, perform analysis on the module graph of the application to ensure that all module dependencies are satisfied.

The `java.base` Module

The `java.base` module implements the *fundamental* APIs of the Java SE platform. It is necessary for any Java program to function, and deemed necessary for every configuration of the Java platform. Not surprisingly, it is required by every module in the JDK. In fact, *every* application module has an *implicit dependence* on the `java.base` module. However, it is seldom necessary for a module to explicitly specify a dependence on the `java.base` module.

Typically when constructing a module graph, dependency on the `java.base` module is only shown for a module that does not depend on any other module. Otherwise, this dependency is implicit.

The `java.base` module contains many important packages, such as `java.lang`, `java.util`, and `java.io`. The `java.lang` package, which contains the primordial class `java.lang.Object`, is imported automatically by every Java compilation unit.

Even though every module depends on the `java.base` module, this does not mean that all packages in it are also automatically *imported*. This only applies to the `java.lang` package. To use simple names of types in the source code, the relevant packages must be explicitly imported.

The `java.se` Module

The `java.se` module represents the *core* Java platform. It depends on each module in the Java SE platform configuration. Technically called an *aggregate module*, it does not contain any packages, but acts as a conduit to channel dependencies on other modules. Any module that depends on the `java.se` module would effectively imply that it depends on all modules of the Java SE platform. A typical scenario is for an application to depend on `java.base` to start with, and add dependencies to other modules in the JDK as and when needed.

19.3 Module Basics

Underlying the Java Module System is the concept of a *module* which is defined in the language and supported by the various JDK tools (such as `javac`, `java`, `jar`, and `jdeps`) and the Java runtime environment.

An understanding of packages is a prerequisite for understanding modules. The discussion of packages in §6.3, p. 326, is therefore highly relevant to this chapter. Going forward, we assume that creating and using packages in Java is well understood.

Module Declaration

A *module declaration* embodies the definition of a module that contains packages— that is, it specifies a module's properties. A module declaration must specify the *name* of the module and can declare the following in its declaration:

- *Dependencies* the module has on other modules—that is, modules it *requires* to implement its functionality
- Packages it *exports* for other modules to use
- Packages it *opens* for *reflection* to other modules (p. 1191)
- Services it *provides* (p. 1196)
- Services it *consumes* (p. 1196)

In essence, a modular declaration specifies any modules its packages (and hence their classes) require to implement their functionality, and any packages that are provided for the benefit of other modules. Unless otherwise specified, other packages in the module are internal to the module—that is, they are not accessible to other modules.

The *basic form* of a module declaration is shown below. A module has a unique *name*. The *body* of the modular declaration can include *module directives*. The `requires` and `exports` *directives* are shown in the module declaration below. The modules required by a module for its internal implementation are specified using a `requires` directive, and the packages that other modules can utilize from it are specified using an `exports` directive. The latter is called an *unqualified export* as the exported package is available to any module that requires this module. (See §19.4, p. 1177, for an overview of all module directives.)

Click here to view code image

```
module moduleName {
  // Body:
  requires moduleName_1;           // A requires directive.
  ...
  requires moduleName_n;
  exports packageName_1;           // An exports directive.
  ...
  exports packageName_m;
}
```

There can be zero or more such directives in such a module declaration. So a module declaration with no directive is valid, as it neither requires any modules (bar the `java.base` module) nor exports any of its packages.

The same directive cannot be duplicated. The directives can occur in any order, but conventionally they are grouped with `requires` directives preceding `exports` directives. In a module declaration, the word `module` is treated as a keyword only in the context of specifying the module name, and the words `requires` and `exports` are treated as keywords only in the context of specifying a module directive. They can be used as normal identifiers in all other contexts.

Only one module declaration can be specified in a file, which must be named `module-info.java`. No other declarations can be included in this file, and there can be no `package` statement either, but the `import` statement is allowed. This file also cannot be empty. As any other Java source code file, the `module-info.java` file with the module declaration is compiled and its bytecode is placed in a class file named `module-info.class`, called the *module descriptor*.

The graphical notation in Figure 19.3(a) and (b) shows two modules, `client` and `seller`, with their corresponding module declarations in Figure 19.3(c) and (d), respectively.

Figure 19.3 *Modules and Module Declarations*

- *The name of the module*: This is the unique *name* of the module to distinguish it from other modules.

In Figure 19.3(a) and (b), the names of the modules are `client` and `seller`, respectively. These module names are used in the respective module declaration in Figure 19.3(c) and (d).

- *The required modules*: These are the modules that a module *requires* for the types (i.e., classes and interfaces) in its packages.

  In Figure 19.3(a) and (b), the `client` module requires the `seller` module and the `seller` module does not require any module, respectively. This is reflected in the respective module declarations in Figure 19.3(c) and (d): The module declaration for `client` has a `requires` directive specifying the `seller` module, but the module declaration for `seller` has no `requires` directive. The `requires` directive states that the specified module is required at compile time and runtime by the declared module.

  The `client` module has a dependence on the `seller` module, called a *module dependency*, shown with a directed arrow from `client` module to `seller` module in Figure 19.3. Technically, the `client` module is said *read* the `seller` module. Often, we say module `client` *depends on* module `seller`, or module `client` *requires* module `seller`, or module `seller` *is readable* by module `client`. Although technically not correct, for the purpose of this discussion, they are all synonyms.

- *The exported packages*: These are packages that a module *exports* so other modules can utilize them for their own implementation. Exported packages define the *public API* of the declared module, where *accessibility rules* govern what types (and hence which members in these types) in an exported package are accessible to other modules (p. 1174).

  In Figure 19.3(a) and (b), the `client` module does not export any packages and the `seller` module exports packages `com.abc.seller` and `com.abc.factory`. This is reflected in the respective module declarations: The module declaration for `client` has no `exports` directive, but the module declaration for `seller` has two `exports` directives that specify the exported packages.

  Although the package `com.abc.factory` in module `seller` is exported, it does *not* imply that the package `com.abc.factory.test` is exported as well. This is analogous to the `import` statement, where packages must be imported individually.

- *The internal packages*: These are the packages that are *internal* to the module and therefore not accessible to other modules.

  In Figure 19.3(a) and (b), the `client` module has an internal package `com.abc.client` and the `seller` module has an internal package `com.abc.factory.test`. However, there is no directive to specify an internal package in the module declaration. The question of how we implement a module and its packages in the file system is answered in §19.5, p. 1179.

Module Name

A legal module name can consist of one or more (by convention, lowercase) legal identifiers separated by a `.` (dot), as in the case of a package name. Module names and package names are in different *namespaces*. From the context in which such a name occurs, the compiler can deduce which namespace it belongs to. For example, the name `com.abc.seller` is a module name if used with the context-dependent keyword `module` in a module declaration, but it is a package name when used in an `exports` directive.

Analogous to package names, reverse DNS (*Domain Name System*) notation can be used to form unique module names. The convention is to use the name of the *principal exported package* (if there is one) in a module as its module name. When naming a module, it is recommended that the name should reflect the *structure* of the module—that is, its package hierarchy. For example, the module declarations in Figure 19.3(c) and (d) can be written as follows, respectively:

Click here to view code image

```
module com.abc.client {            module com.abc.seller {
  requires com.abc.seller;           exports com.abc.seller;
}                                    exports com.abc.factory;
                                   }
```

If care is not exercised in naming modules and packages, distinguishing between module names and package names can become a problem when reading the code, as we can see in the module declarations above, where the name `com.abc.seller` is used both as a module name and as a package name. For brevity, we will not adhere to the naming scheme described above and will continue to use simple names for modules, as in Figure 19.3.

A package is implemented in the file system by a directory hierarchy that mirrors its name. For example, a package named `com.abc.seller` is implemented by its source code files being located under the directory `./com/abc/seller`. As we shall see in §19.5, p. 1181, a module named `com.abc.seller` is implemented by a directory whose name is `com.abc.seller` under which all its packages are located.

Some examples of legal module names (as well as legal package names):

Click here to view code image

```
my.big.fat.module, com.passion.controller, org.geek.gui, view
```

Some examples of illegal module names (as well as illegal package names):

Click here to view code image

```
net.soda.7up, org:factory, com-factory-gizmo
```

Module Graph

The structure of an application is reflected by its module graph constructed from dependencies between its modules. An important aspect of the module system is to perform *module resolution* at compile time and runtime to ensure that the application has a *reliable configuration*. This process *resolves* all dependencies, making sure that they are satisfied—that is, the `requires` directives are matched by `exports` directives with all types required being accessible.

The module resolution mandates that the module graph is *acyclic*, meaning that the graph cannot have *dependency cycles* where a module depends on itself, either directly or indirectly. The module system also does not allow a package to be split between different modules, primarily because this would mean searching in several modules to locate a type in a split package. Therefore, all code in a package must be in the same module.

Every module depends on the proverbial `java.base` module in the JDK. Its packages are available to all modules—for example, the package `java.lang` containing classes like `Object` and `String`. Specifying `java.base` in a `requires` directive is optional in a module declaration. If not specified explicitly, an *implicit dependency* on `java.base` is automatically applied.

Figure 19.4(a) shows the module graph from Figure 19.3 augmented by the two dependencies on `java.base`. The declarations of module `client` and module `seller` in Figure 19.4(b) and (c) include explicit `requires` directives specifying `java.base`. The convention is to leave the dependency on `java.base` implicit. Showing all implicit dependencies from all modules on `java.base` in a module graph can quickly make the graph look like spaghetti. Often this dependency is only shown for modules that do not depend on other modules (a.k.a. *sink nodes* in graph terminology).

Figure 19.4 *Module Graph and Implicit Dependencies*

Readability and Accessibility

A *readability* relationship is implied by the `requires` directive. Module `client` *reads* module `seller`, since the module declaration of `client` specifies the following `requires` directive:

Click here to view code image

```
requires seller;                                 // (1)
```

This `requires` directive alone is not enough to allow code in module `client` to access types in packages contained in module `seller`. Module `seller` must reciprocally *export* the relevant packages required by module `client`, as in the following `exports` directive in the declaration of module `seller`:

Click here to view code image

```
exports com.abc.seller;                          // (2)
```

What if the code in module `client` wants to access the class `GizmoSeller` in the package `com.abc.seller` of module `seller`? In this case, this class is only accessible to module `client` if it is declared `public` in the package `com.abc.seller`:

Click here to view code image

```
public class GizmoSeller { /*...*/ }             // (3)
```

The three conditions, as exemplified by (1), (2), and (3) above, must be satisfied for the accessibility relationship. The module system ensures, both at compile time and runtime, that these conditions are met before allowing access to code from other modules.

We see that only a `public` type in an exported package of a module is accessible by modules that read this module. The exported `public` types comprise the *public API* of a module.

Accessibility Rules for Members

Table 19.1 summarizes the accessibility rules for *members* of a `public` class `A` contained in a package `pkg1` of module `M`. Note that class `A` is `public` so that it satisfies one of the three conditions discussed earlier for accessibility of a type from another module.

Table 19.1 *Module Accessibility Rules*

Access modifier for members in `public` class `A` in package `pkg1` of module `M`	**Inside module `M`**	
`public`	Accessible in all packages in module `M`	Accessible in all packages in module `N`, if and only if module `N` *requires* module `M` and module `M` *exports* `pkg1`
`protected`	Accessible in package `pkg1`, and accessible only by subclasses of class `A` in other packages in module `M`	Accessible by subclasses of class `A` in all packages in module `N`, if and only if module `N` *requires* module `M` and module `M` *exports* `pkg1`
package-private	Only accessible in package `pkg1`	Not accessible
`private`	Only accessible in class `A`	Not accessible

In Table 19.1, the middle column shows how the members of `public` class `A` are accessible inside module `M`. Accessibility rules for members that are applied before modules are now only applicable for accessing members inside a module.

In module `N`, the `public` and the `protected` members of `public` class `A` are accessible if and only if class `A` is accessible in module `N`. In order for class `A` to be accessible in module `N`, the conditions for accessibility of types must be satisfied: Module `N` *requires* module `M`, module `M` *exports* package `pkg1`, and class `A` in package `pkg1` is `public`. Not surprisingly, `private` and package-private members of `public` class `A` are not accessible in module `N`.

Implied Module Dependencies

As an example of refactoring modules, we will refactor the `seller` module in Figure 19.3(b). We decide to move the package `com.abc.factory` and its subpackage `com.abc.factory.test` to a new module called `factory`. The effects of this factoring are illustrated in Figure 19.5. Now both `client` and `seller` require `factory`, as can be seen from their module declarations in Figure 19.5(d) and (e). The package `com.abc.factory` is now exported by the module `factory` in Figure 19.5(f). Apart from fixing the module declarations and moving the relevant packages from `seller` to `factory`, no other changes are necessary.

Figure 19.5 *Module Declarations and Module Graph*

The module graph in Figure 19.5 shows the dependencies between the modules after refactoring. Note that there are two paths from `client` to `factory` in the module graph. One might be tempted to think that since `client` requires `seller` and `seller` requires `factory`, then `client` requires `factory` transitively, and we could omit the `requires` directive on `factory` in `client`. However, the compiler will diligently flag an error. Dependencies specified by the `requires` directive are *not* transitive.

The `requires transitive` *Directive*

Implied dependencies can be created by the `requires transitive` directive. This is illustrated in Figure 19.6. If it is the case that all modules that require `seller` will also require `factory`, we can omit the `requires` directive on `factory` in `client` and use a `requires transitive` directive in `seller`. In Figure 19.6, `client` no longer requires `factory`, but `seller` specifies `factory` in a `requires transitive` directive. All changes are confined to the module declarations in Figure 19.6(d) and (e), so no other changes are required in the code. Implications of using a `requires transitive` directive are that any module that requires `seller` now will also implicitly require `factory`. This implied dependency is shown from `client` to `factory` in the module graph in Figure 19.6.

Figure 19.6 *Implied Module Dependencies*

Qualified Export

Any package that is exported by a module is readable by any module that requires its module. However, sometimes it is necessary that only certain modules can access an exported package. This can be achieved by using the `exports-to` directive, as shown below at (1), where `packageOne` in `moduleX` is *only* exported to `moduleA` and `moduleB`:

Click here to view code image

```
module moduleX {
  exports packageOne to moduleA, moduleB;    // (1) Qualified export.
  exports packageTwo;                        // (2) Unqualified export.
}
```

This is known as *qualified export*. Apart from `moduleA` and `moduleB`, no other module can access `packageOne` in `moduleX`. The compiler will issue an error if an attempt is made by any other module. Qualified export does not prevent the modules in the `to` clause to access other packages that are exported. Unqualified export of `packageTwo`, shown at (2), allows *any* module that requires `moduleX` to access `packageTwo`. Qualified export should be used judiciously, as its usage results in tight coupling between modules that can increase the degree of dependency between modules—a hallmark of poor software design.

Figure 19.7 shows how the module declaration of the `factory` module that uses the `exports-to` directive allows its package `com.abc.factory` to be accessible only by the `seller` and the `client` module. Any other module that requires module `factory` will not be able to access the `com.abc.factory` package in the `factory` module.

Figure 19.7 *Qualified Export*

19.4 Overview of Module Directives

An overview of all module directives is given in Table 19.2, together with references to where they are covered in this chapter.

Table 19.2 *Overview of Module Directives*

Directive	Description
`requires` *`module`* *(p. 1170)*	The current module specifies a dependence on the specified *`module`*, allowing the current module to access exported public types in the specified *`module`*.
`requires transitive` *`module`* *(p. 1175)*	Any module that requires the current module will have an implicit dependence on the specified *`module`*, and can access exported public types in the specified *`module`* on par with the current module.
`requires static` *`module`*	The current module has a mandatory dependence on the specified *`module`* at compile time, but is optional at runtime. This is known as *optional dependency*. It is not discussed in this book.
`exports` *`package`* *(p. 1170)*	The specified *`package`* exported by the current module is available to other modules that require the current module—that is, they can access public types in the specified *`package`*. This is called an

	unqualified export.
`exports` *`package`* `to` *`module1`*`, ...` (p. 1176)	The specified *`package`* exported by the current module is solely available to the modules specified in the `to` clause. That is, only those modules in the `to` clause that require the current module can access public types in the specified *`package`*. This is called a *qualified export.*
`opens` *`package`* (p. 1191)	Access to the specified *`package`* is only granted through *reflection* (§25.6, p. 1587) at runtime for code in other modules. This is called an *unqualified opens directive.*
`opens` *`package`* `to` *`module1`*`, ...` (p. 1195)	Access to the specified *`package`* is only granted through *reflection* at runtime solely for code in modules specified in the `to` clause. This is called a *qualified opens directive.*
`provides` *`type`* `with` *`type1`*`, ...` (p. 1196)	The current module specifies that it provides a *service* (specified by *`type`*) that is implemented by one or more *service providers* (*`type1`*`, ...` in the `with` clause implement the service). The current module is said to be a *service provider.* The service provided can be consumed by other modules that want to use this service.
`uses` *`type`* (p. 1196)	The current module specifies that it uses a *service* (specified by *`type`*) and that a *service provider* that implements it may be discoverable at runtime. Typically, the current module is said to be a *service locator*, but can also be a *service consumer* if the locator is merged with the consumer.

Modules add 10 restricted keywords to the language: `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses`, and `with`. The words `open` (see Table 19.3) and `module` are keywords only in the header of a module declaration, and the remaining in a particular context of a specific directive. The word `static` is already a keyword, but has a different meaning in the `requires static` directive.

Table 19.3 ***The* `open` *Modifier for Modules***

Modifier	**Description**
`open module` *`module`* `{ ... }` (p. 1196)	Unrestricted access to *all* packages in the *module* is granted through *reflection* at runtime for code in other modules.

19.5 Creating a Modular Application

After understanding the basic role of a module declaration, we can proceed to create a modular application. As support for Java modules is not quite up to par at present in integrated development environments (IDEs), we will use the command-line tools provided by the JDK. Using the command-line tools also provides a deeper understanding of the Java Module System. We will be using the following JDK tools in the rest of this chapter:

- *Java language compiler: the* `javac` *command (p. 1186)*
- *Java application launcher: the* `java` *command (p. 1186)*
- *Java Archive tool: the* `jar` *command (p. 1189)*
- *Java Class Dependency Analyzer tool: the* `jdeps` *command (p. 1214)*
- *Java Linker tool: the* `jlink` *command (p. 1204)*

The JDK tools are versatile tools with numerous features and many capabilities. The myriad of options for each tool can be overwhelming. Tables with summaries of selected options for each tool are provided, starting with Table 19.6 on page 1218.

As a running example, we will develop an elementary multi-module application that is based on a very simplified Model-View-Controller (MVC) design pattern. Prior knowledge of the MVC design pattern is not essential. To keep things manageable, the emphasis is more on the mechanics of creating a modular application than on providing an industrial-strength implementation of the MVC design pattern.

To make it less tedious in experimenting with the modular application code, a file with the relevant commands is provided, together with the source code of the application, all of which can readily be downloaded from the book’s website. Hands-on coding of a modular application is highly encouraged, to understand both the Java modules and the tools essential for creating modular applications.

Running Example: Dispensing Canned Advice

In the MVC design pattern, the controller handles the user requests. The model provides the business logic and manages the data in the application. The view creates the response to the user, usually by updating the data display. Simplified, the interaction in an MVC-based application proceeds as follows:

- A user sends a request to the controller.
- The controller interacts with the model and with the view.

First it requests the model to update its data according to the user request, and then it requests the view to create a response. The view accesses the model and updates the data display accordingly.

The dispensing canned advice application (called `adviceApp`) embodies the simplified MVC design pattern outlined above. It comprises four modules: the `model`, `view`, `controller`, and `main` modules, shown in Figure 19.8. The `main` module in Figure 19.8 simulates the user mentioned above. From the notation used for a module in Figure 19.8, we can read the name of the module, the modules it requires, the packages it exports, and any packages that are internal to the module. Based on the modules required by each module, the resulting *module graph* is also shown in Figure 19.8, with *dependencies* between them. For example, the `controller` module requires the modules `view` and `model`, thus there are two dependencies from the `controller` module to the `view` and the `model` modules, respectively.

Figure 19.8 *Module Graph of the* `adviceApp` *Application*

Creating the Application Directory Structure

Typically, the code for a modular application is organized in a *directory structure*, similar to the one shown in Figure 19.9. The modules are created in the `src` directory and compiled into the `mods` directory. The compiled modules are bundled into modular JARs and placed in the `mlib` directory. The different JDK tools can thus operate on the relevant directories.

Figure 19.9 *Application Directory Structure*

The application root directory for the `adviceApp` application bears the same name. It is also the *working directory* in which all commands are issued to the JDK tools. Under the `adviceApp` directory, the following command creates the directories shown in Figure 19.9:

```
>mkdir mlib mods src
```

Creating an Exploded Module Directory

The `model` module does not require any user-defined modules—that is, it does not depend on any user-defined module, and can therefore be implemented first. Its module declaration is straightforward, as it exports its only package, `com.passion.model` (shown in Figure 19.10). The directory layout for the `model` module is created manually under the `src` directory as shown in Figure 19.10, and referred to as the *exploded module*.

Figure 19.10 *Exploded Module Structure for the* `model` *Module*

Referring to Figure 19.10, (1) shows that the module name and that of the *module root directory* must be the same, and (2) shows that the `module-info.java` file containing the module declaration must reside immediately under the module root directory.

Packages were discussed in §6.3, p. 326. Any package (or subpackage) that a module contains is mapped to a corresponding directory hierarchy under the module root directory. In Figure 19.10, (3) shows how the package `com.passion.model` of the module is laid out under the module root directory, with the directory hierarchy `com/passion/model` mirroring its package name. It also shows that the package has only one source file, `AdviceModel.java`. If the package had any other source files, they would also be placed in the same location.

A package is included in a module by virtue of its location in the module root directory. The organization of the module root directory, as in Figure 19.10, defines what constitutes a module. Similarly, any other packages in the module are mapped to their directory structure under the module root directory.

Figure 19.11 shows the module graph of the `adviceApp` application together with the module declaration of each module that is in the module graph. Each module declaration is in a source file named `module-info.java`, and is placed in its corresponding exploded module directory, as shown in Figure 19.12.

Figure 19.11 *Module Graph and Module Declarations*

Figure 19.12 *Compiling Exploded Modules Containing Source Code*

Figure 19.12(a) shows the contents of the `src` directory where each module is mapped to its corresponding exploded module. Each module has one package that contains one source file. This package is mapped to its directory hierarchy under the root directory of its module.

Creating the Source Files in the Exploded Modules

The classes that are in packages included in the modules are declared in their respective source files. Example 19.1 shows the source files for the `adviceApp` application.

The source code of the `AdviceModel.java` file in the `model` module is shown at (1) in Example 19.1. This source file should reside in the package `com.passion.model` in the `model` module. Its file path `src/model/com/passion/model/AdviceModel.java` in the exploded module is shown as a comment at (1). The class `AdviceModel` is declared to be in package `com.passion.model`. It keeps track of the current advice (the `String` field `currentAdvice`). Its constructor creates an `AdviceModel` with the field `current-Advice` initialized to the string `"No advice."` The method `setCurrentAdvice()` sets the field `currentAdvice` to the advice corresponding to its `int` parameter value. The method `getCurrentAdvice()` returns the value of the field `currentAdvice`.

The source code of the `AdviceView.java` file in the `view` module is shown at (2) in Example 19.1. This source file should reside in the package `com.passion.view` in the `view` module. Its file path `src/view/com/passion/view/AdviceView.java` in the exploded module is shown as a comment at (2). The class `AdviceView` is declared to be in package `com.passion.view`. It uses an `AdviceModel`. This model is passed to an `AdviceView` in the constructor. The method `updateAdviceDisplay()` accesses its model for the current advice and prints it.

Analogously, the source code of the `AdviceController.java` file in the `controller` module is shown at (3) in Example 19.1. An `AdviceController` uses an `AdviceModel` and an `AdviceView`, both of which are created in the constructor, where the `Advice-Model` is injected into the `AdviceView`. The method `showAdvice()` calls the `AdviceModel` to set the current advice based on its `int` parameter value, followed by a call to the `AdviceView` to update the advice display.

Finally, the source code of the `Main.java` file in the `main` module is shown at (4) in Example 19.1. The `Main` class is the entry point of the application. It simulates a user by calling an `AdviceController` several times to show different advice.

Note that all source code should be explicitly contained in packages, except for the module declaration—that is, the `module-info.java` file, which is always located immediately under the module root directory. The name of the module in the module declaration must match the name of the module root directory.

Example 19.1 *Source Code Files for the* `adviceApp` *Application*

Click here to view code image

```
// File path: src/model/com/passion/model/AdviceModel.java                  (1
package com.passion.model;

public class AdviceModel {

  private String currentAdvice;

  public AdviceModel() { this.setCurrentAdvice(0); }

  public void setCurrentAdvice(int i) {
    String advice;
    switch(i) {
      case 1 : advice = "See no evil."; break;
      case 2 : advice = "Speak no evil."; break;
      case 3 : advice = "Hear no evil."; break;
      default: advice = "No advice.";
    }
    currentAdvice = advice;
  }

  public String getCurrentAdvice() { return currentAdvice; }
}
```

Click here to view code image

```
// File path: src/view/com/passion/view/AdviceView.java                     (2
package com.passion.view;
import com.passion.model.AdviceModel;             // From model module.

public class AdviceView {

  private AdviceModel model;

  public AdviceView(AdviceModel model) { this.model = model; }

  public void updateAdviceDisplay(){
    System.out.println(model.getCurrentAdvice());
  }
}
```

Click here to view code image

```
// File path: src/controller/com/passion/controller/AdviceController.java   (3
package com.passion.controller;
import com.passion.model.AdviceModel;               // From model module.
import com.passion.view.AdviceView;                 // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);                   // Inject the model.
  }

  public void showAdvice(int adviceNumber) {
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

Click here to view code image

```
// File path: src/main/com/passion/main/Main.java                           (4
package com.passion.main;
```

```
import com.passion.controller.AdviceController;    // From controller module.

public class Main {
  public static void main(String... args) {
    AdviceController controller = new AdviceController();
    controller.showAdvice(1);
    controller.showAdvice(2);
    controller.showAdvice(3);
    controller.showAdvice(0);
  }
}
```

19.6 Compiling and Running a Modular Application

Both the `javac` tool to compile Java source code and the `java` tool to launch an application include new command-line options to specifically support building of modular applications.

When the `javac` tool or the `java` tool is called, initially a *module resolution* is performed on the application's structure. This process checks that dependencies among the modules are resolved in the application, thus ensuring a reliable configuration, by catching any problems as early as possible.

The astute reader will soon notice that the tool commands in this chapter are run on a Unix-based platform. Platform-dependent idiosyncrasies are pointed out where appropriate. The first one to note is the line-continuation character to break up a command line over several lines. This is to make the command and its options easier to read. The *line-continuation character* is a `\` (backslash) on the Unix-based platform and a `^` (caret) on the Windows platform. Below, the command on a single line and the command that spans over two lines are equivalent.

```
>javac One.java Two.java

>javac One.java\
 Two.java
```

Individual Module Compilation

We first look at how to compile the modules of the `adviceApp` application individually. Typically, in order to compile a module, the compiler needs to know the source directory to find the source code of the module, the destination directory to place the Java bytecode class files for the module, and any additional modules that the module requires.

The modules in an application can only be compiled in an order that respects the module dependencies—that is, all modules a module requires have already been compiled—or if it does not depend on any other modules. In the case of the `advice-App` application, there is only one such order, shown in Figure 19.12(b). We can compile the `model` module first since it does not require any other modules (apart from the `java.base` module that is readily accessible by default). Figure 19.13 shows how the `model` module is compiled. The `javac` command in Figure 19.13(c) shows how the source code of the `model` module under the `src/model` directory is compiled to the `mods/model` directory where the necessary package is created and the class files are placed. Note how the `mods/model` directory mirrors the `src/model` directory.

Figure 19.13 *Compiling the* `model` *Module*

Click here to view code image

```
>javac --module-path mods \
       -d mods/model \
       src/model/module-info.java \
       src/model/com/passion/model/AdviceModel.java
```

The `--module-path` option (*short form*: `-p`) is used to specify where to find the modules required to compile the source code files specified in the command line. In the case of the `model` module, this option is superfluous as this module does not depend on any other module. However, the other modules in the `adviceApp` application do so. As modules are compiled, their exploded modules containing the class files will be readily found in the `mods` directory specified by this option. This option can also specify a colon-separated (`:`) list of directories (semicolon (`;`) separated in Windows) to look up required modules in different directories.

The `-d` option (*no long form*) specifies the *destination directory* (also called *target directory*) for placing the Java bytecode class files. In this case they will be placed under the `mods/model` directory, which is created by the compiler if necessary.

The source files in the module are specified *last* in the command line with their full pathname. The compiler creates the necessary directory hierarchy under the destination directory `mods/model` specified by the `-d` option. The module declaration in the `module-info.java` file is also compiled to a class file, as any other Java source file. This class file is called the *module descriptor*. It resides immediately under the module root directory, which in this case is `mods/model`.

After the `model` module is compiled, the next candidate to compile is the `view` model, as its required module `model` has now been compiled to the `mods/model` directory. Figure 19.14(a) shows the final result of compiling the modules in the `adviceApp` application by executing the `javac` commands in the order shown in Figure 19.12(b).

Figure 19.14 *Creating Modular JARs*

Multi-Module Compilation

It is tedious compiling modules individually. It is more convenient to use the following command:

Click here to view code image

```
>javac --module-source-path src -d mods --module main
```

The `--module-source-path` option (*no short form*), as the name suggests, indicates the `src` directory where the exploded modules with the source files for the modules can be found.

The `-d` option (*no short form*) indicates the directory `mods` for the exploded modules with the class files. The compiler creates the module root directories and the package hierarchies as necessary under the `mods` directory.

The `--module` option (*short form*: `-m`) can be used to specify the name of a single module or a list of module names separated by a comma (`,`) with no intervening space characters. For each module specified, the compiler will only compile those modules that this module depends on. The compiler does the necessary module dependency analysis and compiles all the modules in the right order. If only the `model` module had been specified, only it will be compiled, as it does not depend on any other module.

The result of the above command is the same as before, shown in Figure 19.14(a).

Running a Modular Application from Exploded Modules

Typically, the `java` command needs to know where to find the modules needed to run the application and also the entry point of the application, meaning the class whose `main()` method should be executed to launch the application.

Click here to view code image

```
>java --module-path mods --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

The `--module-path` option (*short form*: `-p`) is used to specify where to find the modules needed to run the application. In this case it is the `mods` directory containing the exploded modules with the class files. This option can also specify a colon-separated (`:`) list of directories (semicolon (`;`) separated in Windows) to look up the required modules.

The `--module` option (*short form*: `-m`) in the command explicitly specifies the *entry point* of the application. In this case, it is the class `Main` in the `com.passion.main` package of the `main` module that provides the `main()` method. This requires the *fully qualified name* of the class (`com.passion.main.Main`) and the *full name* of the module (`main`). Note the *directory-level separator* (`/`) in the specification of the entry point.

19.7 Creating JAR Files

A *JAR* (Java Archive) file is a convenient way of bundling and deploying Java executable code and any other resources that are required (e.g., image or audio files). A JAR file is created by using the `jar` tool. The `jar` command has many options, akin to the Unix `tar` command (p. 1221).

In addition to creating *plain JARs* (also referred to as *non-modular JARs*), the `jar` tool of the JDK has been enhanced to create and manipulate *modular JARs*, making it convenient to bundle and deploy modules. The procedure for creating a plain or a modular JAR is the same; the main difference being that a modular JAR always contains the module descriptor—that is, the `module-info.class` file.

Creating Modular JARs

Figure 19.14(b) shows the `jar` commands that can be used to create a modular JAR for each of the modules in the `adviceApp` application. The resulting modular JARs are placed under the `mlib` directory, as shown in Figure 19.14(c). We take a closer look at the `jar` command to create a modular JAR for the `model` module:

```
>jar --verbose \
     --create \
     --file mlib/model.jar \
     -C mods/model  .
```

Possible output to the terminal:

Click here to view code image

```
added manifest
added module-info: module-info.class
adding: com/(in = 0)  (mods= 0)(stored 0%)
adding: com/passion/(in = 0)  (mods= 0)(stored 0%)
adding: com/passion/model/(in = 0)  (mods= 0)(stored 0%)
adding: com/passion/model/AdviceModel.class(in = 641)  (mods= 415)(deflated 35%
```

The `--verbose` option (*short form*: `-v`) instructs the `jar` tool to print information about its operation on the terminal, as evident from the output on the terminal.

The `--create` option (*short form*: `-c`) results in a modular JAR to be created, as the module descriptor will be included in the JAR file (see below).

The `--file` option (*short form*: `-f`) specifies the file name of the modular JAR. In this case, it is `model.jar` that will be created and located under the `mlib` directory. Note there are no restrictions on the file name of the modular JAR, as long as it is a legal file name. Of course, having a descriptive file name that reflects what the module in the modular JAR stands for can aid readability.

The `-C` option instructs the `jar` tool to change to the `mods/model` directory and include the contents of this directory which is designated by the dot (`.`) special notation. From the output on the terminal we can see what is added to the JAR file. Note that the modular JAR does not include information about the module root directory (in the above case, `mods/model`) in the modular JAR because the module descriptor already has information about the name of the module. The contents included in the modular JARs are shown by the dashed boxes in Figure 19.14(a).

The observant reader will notice that the `jar` command in Figure 19.14(b) to create the `main.jar` has an extra option to specify the application entry point:

Click here to view code image

```
>jar --verbose \
     --create \
     --file mlib/main.jar \
        --main-class com.passion.main.Main \
        -C mods/main .
```

Since the entry point of the application is in the `model` module, the `--main-class` option (*short form*: `-e`) specifies the fully qualified name of the class containing the `main()` method—that is, `com.passion.main.Main`. The module name is *not* specified.

Using the short form of the first four options in the above `jar` command, respectively (see Table 19.6), the command can be written as follows:

Click here to view code image

```
>jar -vcfe mlib/main.jar com.passion.main.Main -C mods/main .
```

Note that if the files named `module-info.class` are deleted in Figure 19.14(a), the `jar` commands in Figure 19.14(b) will still create JAR files, but these will be plain JARs, and not modular JARs.

Running an Application from a Modular JAR

The `adviceApp` application can be run from the modular JARs in the `mlib` directory by any of the following `java` commands, giving the same output as before:

Click here to view code image

```
>java --module-path mlib --module main
```

or

Click here to view code image

```
>java --module-path mlib --module main/com.passion.main.Main
```

The `--module-path` option (*short form*: `-p`) specifies the `mlib` directory in which to find the modular JARs to run the application. Previously the exploded module directory `mods` containing the class files for each module was specified.

As seen previously, the `--module` option (*short form*: `-m`) specifies the entry point of the application in the `java` command. In the first `java` command, only the module name `main` is specified. The `main` module can be found in the `main.jar` file under the `mlib` directory by examining the module descriptor in each module on the module path. The `main.jar` file contains the information about the class with the `main()` method, as it was specified when the modular JAR for the `main` module was created—that is, `com.passion.main.Main`. In the second `java` command, the entry point is explicitly given, with the module name and the fully qualified class name. Specifying the entry point explicitly overrides the entry point in any modular JAR. However, it makes no difference which command we use in the example above, as there is only one entry point for the application.

We have seen how to create, compile, bundle, and run a modular application. Now we take up the discussion about the remaining directives that can be declared in a module declaration: opening code for reflection (p. 1191) and providing services (p. 1196).

19.8 Open Modules and the `opens` Directive

Reflection is a powerful feature in Java that enables inspection and modification of runtime behavior of Java programs. The Reflection API is primarily in the `java.lang.reflect` package. Reference types (classes, interfaces, and enums) can be inspected at runtime to access information about the declarations contained in them (e.g., fields, methods, and constructors)—often referred to as *reflective access*. In fact, classes can be programmatically instantiated and methods can be invoked and values of fields can be changed. An example of using reflection to process annotations in Java programs is provided in §25.6, p. 1587.

Some frameworks, such as Spring and Hibernate, make heavy use of reflection in the services they provide. They rely on *deep reflection*: being able to use reflection, not only on `public` classes and their `public` members, but also on non-`public` classes and non-`public` members. Note that these frameworks require access for reflection at runtime, and not at compile time.

However, things changed with the introduction of modules in Java, where encapsulation is in the front seat. Reflection is now only allowed on `public` members of `public` types in *exported* packages from a module. In other words, the `exports` directive does not permit reflection on non-`public` types and non-`public` members from exported packages.

The `opens` directive in a module declaration addresses this issue of reflection at runtime, allowing deep reflection on *all* classes and *all* members contained in an *open package*.

Click here to view code image

```
module org.liberal {
  opens org.liberal.sesame;     // An open package
}
```

The `adviceApp` application (Example 19.1, Figure 19.12) is now modified to illustrate the `opens` directive. We will refer to the modified application as `adviceOpen`. The module `controller` opens the `com.passion.controller` package to enable clients to inspect its class `AdviceController` at runtime, and elicit advice by invoking the method `showAdvice()`. The declaration of the module `controller` specifies this vital information using an `opens` directive, as shown below in the module declaration at (1).

The idea is that module `main` will use reflective access on the `controller` module to elicit advice. Therefore, the `main` module does not *require* the `controller` module, as we can see from the module declaration below at (2).

It is not necessary for a module to explicitly require a module that has open packages for the purposes of reflection. As long as the module with the open packages is available at runtime, reflection can be used to access the contents of the open packages. The emphasis here is on use of the `opens` directive rather than a fullblown treatise on reflection.

Click here to view code image

```
// Filepath: src/controller/module-info.java
module controller {                                          // (1)
  requires model;
  requires view;
  opens com.passion.controller;                             // Open package
}
```

Click here to view code image

```
// Filepath: src/main/module-info.java
module main {}                                               // (2)
```

In Example 19.2, the `main()` method at (1) in the class `Main` of the package `com.passion.main` in the module `main` uses reflection to invoke the `showAdvice()` method of the `AdviceController` class in the `com.passion.controller` open package.

The comments in the `main()` method are self-explanatory, but a few things should be noted.

The `forName()` `static` method of the `Class` class at (2) loads the specified class and returns the `Class` object associated with the class whose name is `com.passion .controller.AdviceController`. Note that module `controller` is not specified. We will make it available when we run the application with the `java` command.

The `Class` object from (2) is used at (3) to obtain the object representing the no-argument constructor of the `AdviceController` class. The `newInstance()` method is invoked at (4) on the constructor object to create an instance of the `AdviceController` class.

The class object from (2) is used at (5) to obtain all `Method` objects that represent methods in the `AdviceController` class. The method `getDeclaredMethods()` returns an array containing `Method` objects reflecting *all* declared methods of the `Advice-Controller` class represented by the `Class` object `cObj`, including `public`, `protected`, default access, and `private` methods in the class. As this class only has one `private` method—`showAdvice()`—index 0 designates this method in the array of `Method` objects, as shown at (6).

The `if` statement at (7) ensures that the `method` reference indeed designates the `private` method `showAdvice()`.

However, before a non-`public` method is invoked via reflection, it is necessary to explicitly suppress checks for language access control at runtime by calling the `setAccessible()` method of the `Method` class with the value `true`, as shown at (8).

At (9) through (12), the `private` method `showAdvice()` designated by the `method` reference is invoked on the `AdviceController` instance specified in the argument to the `invoke()` method, together with an `int` parameter required by the `showAdvice()` method for specific advice.

The code in the `main()` method is enclosed in a `try-catch` construct to catch the checked exceptions that can be thrown by the various methods.

Example 19.2 *Selected Source Code Files in the* `adviceOpen` *Application*

Click here to view code image

```
// File path: src/main/com/passion/main/Main.java
package com.passion.main;
import java.lang.reflect.*;                             // Types for reflectio

public class Main {
  public static void main(String... args) {                                //
```

```
    try {
      // Get the runtime object representing the class.
      Class<?> cObj
          = Class.forName("com.passion.controller.AdviceController");      //

      // Get the no-argument constructor of the class.
      Constructor<?> constructor = cObj.getDeclaredConstructor();          //

      // Create an instance of the class.
      Object instance = constructor.newInstance();                         //

      // Get all declared methods, including those that are non-public.
      Method[] methods = cObj.getDeclaredMethods();                        //
      // Class has only one method.
      Method method = methods[0];                                          //
      // Check if it is the right method.
      if (!method.getName().equals("showAdvice")) {                        //
        System.out.printf("Method showAdvice() not found in %s.%n",
                          cObj.getName());
        return;
      }

      // Disable access control checks on the method as it is a private method
      method.setAccessible(true);                                          //

      // Invoke the method on the instance, passing any arguments.
      method.invoke(instance, 1);                                          //
      method.invoke(instance, 2);                                          //
      method.invoke(instance, 3);                                          //
      method.invoke(instance, 0);                                          //

    } catch (ClassNotFoundException | NoSuchMethodException |
             InstantiationException | IllegalAccessException |
             IllegalArgumentException | InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

Click here to view code image

```
// File path: src/controller/com/passion/controller/AdviceController.java
package com.passion.controller;

import com.passion.model.AdviceModel;               // From model module.
import com.passion.view.AdviceView;                 // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);
  }

  private void showAdvice(int adviceNumber) {       // (13)
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

The accessibility of the `showAdvice()` method at (13) in class `AdviceController` is changed to `private` in Example 19.2 to illustrate that an open package allows reflection on `private` members.

The other two modules, `model` and `view`, are the same as in the `adviceApp` application (Example 19.1, Figure 19.12).

We can compile all modules in the `adviceOpen` application, as we did for the `adviceApp`:

Click here to view code image

```
>javac -d mods --module-source-path src --module model,view,controller,main
```

However, trying to run the application throws an exception:

Click here to view code image

```
>java --module-path mods --module main/com.passion.main.Main
java.lang.ClassNotFoundException: com.passion.controller.AdviceController
...
```

The reason for the exception is that the `controller` module has not been resolved at runtime, since it is not required by any module in the application. We can include modules needed by the application using the `--add-modules` option:

Click here to view code image

```
>java --add-modules controller --module-path mods \
      --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

Qualified Opens Directive

Sometimes it is necessary that only certain modules can access an open package for reflection. This can be achieved by using the `opens-to` directive, as shown below at (1), where package `org.liberal.sesame` in module `org.liberal` is *only* open to modules `com.aladdin` and `forty.thieves`. This is known as a *qualified opens* directive. The `to` clause can be used to specify a comma-separated list of modules that can inspect the open package.

Click here to view code image

```
module org.liberal {
  opens org.liberal.sesame to com.aladdin, forty.thieves; // (1) Qualified ope
  opens org.liberal.pandorasbox;                          // (2) Unqualified o
}
```

Apart from modules `com.aladdin` and `forty.thieves`, no other module can inspect package `org.liberal.sesame` in module `org.liberal`. Any attempt made by other modules to inspect package `org.liberal.sesame` will result in a `java.lang.IllegalAccessException`. Unqualified open package `org.liberal.pandorasbox`, shown at (2), allows *any* module to inspect this package.

The declaration of module `controller` can be modified to use the qualified `opens` directive, as shown below. The application `adviceOpen` can be compiled and run as before.

Click here to view code image

```
module controller {
  requires model;
  requires view;
  opens com.passion.controller to main;    // Package is only open to main modu
}
```

Open Modules

An *open module* can be declared with the restricted keyword `open` in its declaration file `module-info.java`:

Click here to view code image

```
open module module_name {
  // Any number of module directives, except the opens directive.
}
```

An open module does not declare any open packages, since its declaration implies that all its packages are open. However, note that making a module open also means that it would be possible to use reflection on *all types and their members in all packages* in the module. Making a module open should be done judiciously, and in a way that does not weaken the principle of encapsulation.

The declaration of module `controller` can be modified so that the module is open for other modules, as shown below. The application `adviceOpen` can be compiled and run as before.

Click here to view code image

```
open module controller {                  // An open module
  requires model;
  requires view;
}
```

19.9 Services

Programming to interfaces is a powerful design paradigm that advocates writing code using interfaces and abstract classes—that is, using abstract types and not concrete types. This allows new implementations of abstract types to be used by the code if and when necessary. This strategy results in the code being loosely coupled.

The `requires` directives in module declarations create explicit dependencies between concrete modules, requiring explicit naming of modules with heavy reliance on concrete types, thus making the modules tightly coupled. As we shall see, services allow programming to interfaces, but at a higher level of abstraction, resulting in the modules being loosely coupled, as direct dependencies between them are removed.

The main advantage of services is application *extensibility*: adding functionality by providing new services without having to recompile the whole application. This is feasible as services are discoverable at runtime. This design strategy is typically employed in implementing *plugins* which can extend the capabilities of an application.

A *service* is a specific abstract type, typically an interface, but it can be an abstract class as well, that specifies some specific functionality. The following artifacts are typically required to create a service:

- *The service interface*

  The service interface specifies the abstract type that represents the service. It is also known as a *service provider interface*.

- *The service provider*

  A service provider implements the service. There can be zero or more service providers of a service. A service provider advertises the implementation of a specific service with the `provides-with` directive.

- *The service locator*

  The service locator discovers service providers at runtime through a *service loader* which is an instance of the `java.util.ServiceLoader<S>` class, where type parameter `S` designates the type of service to be loaded. A service loader locates and loads service providers at runtime. A service locator announces the services it is interested in with the `uses` directive.

- *The service consumer*

  A service consumer utilizes the service. The service consumer accesses a particular implementation of the service via the *service locator*, and does not communicate directly with a service provider. It can choose which implementation of the service to consume from the ones provided by the service locator.

We illustrate the setup necessary to use services by implementing a variant of the `adviceApp` we have seen earlier (Example 19.1, Figure 19.12). Each artifact required to set up a service will be implemented as a module in our example below, although other configurations are possible, typically where the service locator is merged with either the service interface or the service consumer. The service providers, being autonomous modules, can be added or removed as necessary.

Figure 19.15, p. 1198 shows the module diagram and the module declarations of the `adviceService` application, and Example 19.3 to Example 19.6 show the source code in each module of the application.

Figure 19.15 *Module Graph and Module Declarations for Services*

Specifying the Service Interface

The service is represented by the interface `IAdvice` that is declared in package `org.advice.si` of the `serviceinterface` module (Example 19.3). It specifies the functionality that the service will provide: two abstract methods `getContent()` and `getLocale()` that return the advice (as a string) and the associated locale at (1) and (2), respectively. In other words, the advice service is locale-specific. Since the two methods have no accessibility modifier in their declaration, they are implicitly `public`, and as they do not provide an implementation, they are implicitly `abstract`.

Example 19.3 *Specifying the* `Advice` *Service Interface*

Click here to view code image

```
// File: ./src/serviceinterface/org/advice/si/IAdvice.java
package org.advice.si;
import java.util.Locale;

// Service interface
public interface IAdvice {
  String getContent();                   // (1) Returns the content of the advi
  Locale getLocale();                    // (2) Returns the associated locale.
}
```

The declaration of the `serviceinterface` module in Figure 19.15(c) exports the package `org.advice.si` containing the service interface `IAdvice` so that service providers can implement it.

Click here to view code image

```
// File: ./src/serviceinterface/module-info.java
module serviceinterface {
   exports org.advice.si;
}
```

Implementing Service Providers

Two service providers are implemented in the `serviceprovider` module (Example 19.4). The package `org.advice.serviceprovider` contains the class `Canned-Advice` that implements the service interface `IAdvice`. The `getContent()` and the `getLocale()` methods return the canned advice and the associated UK locale at (1) and (2), respectively.

Example 19.4 *Implementing the Advice Service Providers*

Click here to view code image

```
// File:./src/serviceprovider/org/advice/serviceprovider/CannedAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class CannedAdvice implements IAdvice {
  public String getContent() {                               // (1)
    return "Keep calm and service on!";
  }
  public Locale getLocale() { return Locale.UK; }            // (2)
}
```

Click here to view code image

```
// File:./src/serviceprovider/org/advice/serviceprovider/VikingAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class VikingAdvice implements IAdvice {
  public String getContent() {                                            // (3)
    return "Gi aldri opp!";    // Never give up!
  }
  public Locale getLocale() { return new Locale("no", "Norway"); }  // (4)
}
```

The package `org.advice.serviceprovider` also contains the class

`VikingAdvice` that implements the service interface `IAdvice`. The `getContent()` and the `getLocale()` methods return the Viking advice and the associated Norwegian locale at (3) and (4), respectively.

The `serviceprovider` module in Figure 19.15(d) requires the module `serviceinterface` in order to implement the service. The module advertizes that it implements the service in a `provides-with` directive at (1) below. In the directive, the service interface that is implemented is specified first, and the `with` clause specifies a comma-separated list of *classes* in this module that implement the service. This information is used by the service loader at runtime to locate and load the classes that implement this service in this module. Note that the package implementing the service is not exported, avoiding any service consumers having explicit dependency on service providers.

Click here to view code image

```
// File: ./src/serviceprovider/module-info.java
module serviceprovider {
  requires serviceinterface;

  provides org.advice.si.IAdvice with                    // (1)
    org.advice.serviceprovider.CannedAdvice,
    org.advice.serviceprovider.VikingAdvice;
}
```

Implementing the Service Locator

The class `java.util.ServiceLoader<S>`, where the type parameter `S` designates the type of service, is at the core of finding and loading service providers for the service `S`. A service loader for a specific service is created by calling the static method `load()` of the `ServiceLoader` class, passing the runtime `Class` object of the service interface, as shown below. Typically, a stream is then created by calling the `stream()` method on the service loader. This stream can be used to lazily load the available service providers for the specified service `S`. The type of an element in this stream is the static member interface `ServiceProvider.Provider<S>`.

Click here to view code image

```
ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream();
```

The procedure above allows processing of stream elements as type `Provider<S>` without instantiating the service provider. The `get()`

method of the `Provider<S>` interface must be invoked on a stream element of type `Provider<S>` to obtain an instance of the service provider—that is, the class that implements the service interface `S`. Given the service provider, the service implemented by the provider can be utilized.

> Click here to view code image
>
> ```
> static <S> ServiceLoader<S> load(Class<S> service)
> ```
>
> This static method creates a service loader for the given service type specified by its runtime object, using the current thread's context class loader.
>
> Click here to view code image
>
> ```
> Stream<ServiceLoader.Provider<S>> stream()
> ```
>
> Returns a stream that lazily loads available providers of this loader's service (designated by type parameter `S`). The stream elements are of type `Service-Loader.Provider<S>`.
>
> The interface `Provider<S>` is a `static` member interface of the class `Service-Loader<S>` that extends the `java.util.function.Supplier<S>` functional interface. The `get()` method of the `Provider<S>` interface must be invoked to get or instantiate the service provider for the service `S`.

A service locator for the advice service is implemented by the class `org.advice .servicelocator.AdviceLocator` in the `servicelocator` module (Example 19.5). The class provides two static methods, `getAdvice()` and `getAllAdvice()`, which return a service provider that implements the `IAdvice` service interface for a particular locale or *all* service providers that implement the `IAdvice` service interface, as shown at (1) and (8), respectively. In other words, the former returns a service provider for a particular locale and the latter returns all service providers that implement the `IAdvice` service interface.

The method `getAdvice()` at (1) is passed the desired locale. Its return type is `Optional<IAdvice>`, as there might not be any service provider for this service for the desired locale. At (2), a service loader is created for the `IAdvice` service interface. A stream is built with the service loader as the source at (3). This stream of element type `Provider<IAdvice>` is processed to find a service provider for the desired locale. Note how at (5), the stream is converted from type `Stream<ServiceLoader.Provider<IAdvice>>` to

`Stream<IAdvice>` by the `map()` intermediate operation that applies the `get()` method of the `Provider<S>` interface to each element. The resulting stream is filtered at (6) for a service provider with the desired locale. The first service provider that has the desired locale is selected by the `findFirst()` terminal operation at (7). This operation returns an `Optional<IAdvice>` as there might not be such a service provider in the stream.

Analogously, the method `getAllAdvice()` at (8) collects all service providers for the `IAdvice` service interface in a list and returns them. In this case, an empty list will indicate that no service providers were found.

Example 19.5 *Implementing the Advice Service Locator*

Click here to view code image

```
// File:./src/servicelocator/org/advice/servicelocator/AdviceLocator.java
package org.advice.servicelocator;

import java.util.*;
import java.util.stream.*;
import org.advice.si.*;

public class AdviceLocator {

  /** Get advice for a particular locale. */
  public static Optional<IAdvice> getAdvice(Locale desiredLocale) {      // (1)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);   // (2)
    Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream();  // (3)
    Optional<IAdvice> optAdvice = spStream                               // (4)
      .map(ServiceLoader.Provider::get)                   // (5) Stream<IAdvic
      .filter(a -> desiredLocale.equals(a.getLocale()))   // (6)
      .findFirst();                                       // (7)
    return optAdvice;
  }

  /** Get all advice implemented by all providers. */
  public static List<IAdvice> getAllAdvice() {            // (8)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
    List<IAdvice> allAdvice = loader
      .stream()
      .map(p -> p.get())                  // (9) map(ServiceLoader.Provider::g
      .collect(Collectors.toList());      // (10) List<IAdvice>
    return allAdvice;
  }
}
```

At (1) below, the `servicelocator` module (Figure 19.15(b)) requires the `serviceinterface` module in order to locate its service providers, and it exports the package `org.advice.servicelocator` at (2) so that service consumers can locate service providers. The `uses` directive at (3) declares which service (`org.advice.si.IAdvice`) is used by this module, allowing it to look up this particular service at runtime.

Click here to view code image

```
// File: ./src/servicelocator/module-info.java
module servicelocator {
  requires serviceinterface;           // (1)
  exports org.advice.servicelocator;   // (2)
  uses org.advice.si.IAdvice;          // (3) In module serviceinterface.
}
```

Implementing the Service Consumer

The class `AdviceConsumer` implements a simple service consumer for the `IAdvice` service in the package `org.advice.serviceconsumer` of the `serviceconsumer` module (Example 19.6).

The code at (1) shows how advice for the UK locale is obtained via the `org.advice.servicelocator.AdviceLocator` class in the `servicelocator` module. The static method `getAdvice()` of the `AdviceLocator` class at (2) returns an `Optional<IAdvice>` object. This object can be queried to extract the advice at (3) with the `getContent()` method, if there is one. Otherwise, the `orElse()` method at (4) provides an appropriate message.

The code at (5) shows how all advice providers for the `IAdvice` service can be obtained via the `AdviceLocator` class. The static method `getAllAdvice()` of the `AdviceLocator` class at (6) returns a `List<IAdvice>` object with all providers for this service. A stream on this list is used at (7) to print the advice from each provider by extracting the advice with the `getContent()` method.

Example 19.6 *Implementing of the Advice Service Consumer*

Click here to view code image

```
// File: ./src/serviceconsumer/org/advice/serviceconsumer/AdviceConsumer.java
package org.advice.serviceconsumer;

import java.util.*;
import org.advice.servicelocator.*;
import org.advice.si.*;

public class AdviceConsumer {
  public static void main(String[] args) {

    // Get advice for the UK locale.                                       // (1)
    Optional<IAdvice> optAdvice = AdviceLocator.getAdvice(Locale.UK);    // (2)
    String adviceStr = optAdvice.map(IAdvice::getContent)                  // (3)
                                .orElse("Sorry. No Advice!");             // (4)
    System.out.println("Advice for UK locale: " + adviceStr);

    // Get all implemented advice.                                         // (5)
    System.out.println("Printing all advice:");
    List<IAdvice> allAdvice = AdviceLocator.getAllAdvice();               // (6)
    allAdvice.stream()                                                     // (7)
      .forEach(a -> System.out.println(a.getContent()));
  }
}
```

The `serviceconsumer` module (Figure 19.15(a)) only requires the module that declares the service, as shown at (1), and the module that can locate service providers for this service as at (2). Note that the consumer module does not directly depend on any service providers.

Click here to view code image

```
// File: ./src/serviceconsumer/module-info.java
module serviceconsumer {
  requires serviceinterface;     // (1)
  requires servicelocator;       // (2)
}
```

Compiling and Running a Service

The directory structure of the `adviceService` application is analogous to that of the `adviceApp` (Figure 19.9, p. 1181).

We can compile all modules in the `adviceService` application as before. However, note that compilation of the `serviceprovider` module does not require any other modules except the `serviceinterface` module.

Click here to view code image

```
>javac -d mods \
   --module-source-path src \
   --module serviceprovider,serviceconsumer,servicelocator,serviceinterface
```

We can run the application as before:

Click here to view code image

```
>java --module-path mods \
      --module serviceconsumer/org.advice.serviceconsumer.AdviceConsumer
Advice for UK locale: Keep calm and service on!
Printing all advice:
Keep calm and service on!
Gi aldri opp!
```

We leave it as an exercise to experiment other configurations of implementing services. In particular, merging the service locator with the service consumer is highly recommended.

19.10 Creating Runtime Images

The *Java Linker* tool, `jlink`, can be used to assemble a customized *runtime image* of a modular application. This image only includes the modules in the application and their dependencies, together with a minimal JRE (Java Runtime Environment) to run the application.

We only present the basic use of the `jlink` tool, and strongly encourage consulting the documentation for the JDK tools for more in-depth coverage.

At a minimum, the `jlink` tool requires the following information to create a runtime image:

- The *module path* given by the `--module-path` option (*short form*: `-p`) to find modules that are to be included in the runtime image.
- The *names of the modules* that should be included in the runtime image. These modules are specified using the `--add-modules` option, and are looked up in the module path. If any standard or JDK modules required by the application are not specified, they are automatically included by the tool. The dependencies between the modules are automatically resolved.
- The *output directory* in which to create the runtime image, specified with the `--output` option.

The following command creates a runtime image of the `adviceApp` in the `advice` directory, assuming that the JAR files for the application modules can be found in the directory `mlib`:

Click here to view code image

```
jlink --module-path mlib --add-modules model,view,controller,main,java.base \
      --output advice
```

The standard module `java.base` is explicitly specified. If omitted, it will be automatically included by the `jlink` tool.

The output directory `advice` containing the runtime image has the following structure, shown below with some of the files in the runtime image. The size of this particular runtime image is approximately 39 MB. Note the `java` command to launch the application under the directory `advice/bin`.

The modules in the runtime image `advice` can be listed by the following command:

Click here to view code image

```
>advice/bin/java --list-modules
controller
java.base@17.0.2
main
model
view
```

The application can be run from the runtime image by the following command, as the entry point of the application is specified in the `main` modular JAR included in the runtime image:

Click here to view code image

```
>advice/bin/java --module main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

A runtime image is platform specific, and thus meant to be run on the platform it was created for. As the runtime image contains platform-specific files, it will most likely not run on other platforms.

It is not possible to automatically update a runtime image with a newer version of Java. The runtime image must be rebuilt with the appropriate version of Java.

Runtime images are beneficial in many ways: They have a smaller memory footprint, can boost performance, are more secure, and are easier to maintain. Not surprisingly, runtime images are ideal for running on smaller devices.

19.11 Categories of Modules

The Java Module System is designed to allow both non-modular and modular code to work together. Types are usually bundled in JAR files. Regardless of whether it a plain JAR or a modular JAR, how its content is handled by the module system depends on the path on which it is placed when using the JDK tools: *the class path* (§6.4, p. 337) or *the module path* (p. 1186). Many JDK tools, like `javac` and `java` commands, allow both paths to be specified on the command line in order to mix non-modular and modular code. Interpretation of JARs placed on either of these two paths is shown in Table 19.4, and is the subject of this section. Understanding this distinction is also essential for migrating code into modules (p. 1209).

Table 19.4 *How JARs Are Handled Depends on the Path*

Path	Plain JAR	Modular JAR
On the class path	Included in *the unnamed module*	Included in *the unnamed module*
On the module path	Treated as an *automatic module*	Treated as an *explicit module*

Searching for types on the class path is different from searching for types on the module path. Class path search is a linear search on the entire class path, finding the first occurrence of a type and terminating if the type is found. Module path search is according to the module dependencies and therefore more efficient.

The Unnamed Module

Types from any JAR that is placed on the class path are included in *the unnamed module*, regardless of whether it is a plain or a modular JAR. If it is a modular JAR, its module descriptor is ignored. However, note that a unique unnamed module is associated with each class loader. Classes loaded by a class loader via the class-path are members of the unnamed module associated with that class loader. Colloquially we refer to *the unnamed module* as the one associated with the application class loader.

The unnamed module is a catchall solution to capture code that is on the class path. It is analogous to code not declared in packages being part of the unnamed package. It is a compatibility feature that allows code on the class path to work with modules on the module path.

Types in the unnamed module can access code from both the class path and the module path—that is, the unnamed module can read *all* modules.

All packages in the unnamed module are exported and open for reflection. Types in the unnamed module are accessible to other types in the unnamed module, and also accessible to *automatic* modules, but only by reflection to explicit modules. An explicit module on the module path cannot access code in the unnamed module, as the unnamed module cannot be specified in a `requires` directive because it has no name. Access to types in the unnamed module is governed by accessibility rules for types in packages—that is, only `public` types in a package are accessible to other packages in the unnamed module and in any automatic modules.

Automatic Modules

A plain JAR—that is, a JAR that does not have a `module-`

`info.class` file in its top-level directory—defines an *automatic module* when placed on the module path. An automatic module has a module name which is determined according to the scheme described below.

An automatic module can read all other modules, including the unnamed module. This means that if an explicit module reads a single automatic module, all other automatic modules can be implicitly read by the explicit module. This avoids having a `requires` directive to every automatic module required by a named module.

All packages in the automatic module are exported and open for reflection. Access to its types is solely determined by the accessibility rules for types in packages. Explicit modules can read automatic modules by specifying them in the `requires` directives, as they have a name.

An automatic module is a migration feature. Resorting to an automatic module should be a temporary measure. As such, a module should be migrated to an explicit module by defining an appropriate module descriptor.

Scheme for Naming Automatic Modules

If the JAR file has the attribute `Automatic-Module-Name` defined in its `META-INF/ MANIFEST.MF` file (p. 1212), its value is used as the module name. This is the recommended practice, as it avoids issues like name clashes.

If the JAR file does not have the attribute `Automatic-Module-Name` defined in its `META-INF/MANIFEST.MF` file, its name is derived from the *file name* of the JAR file according to the following algorithm:

- Remove the `.jar` suffix from the name.
- Remove any version information at the end of the name, which is usually specified in the version number format after a hyphen (`-`); for example, `-1.2.3-ERC`.
- All non-alphanumeric characters are replaced with a dot (`.`), any sequence of dots is replaced with a single dot, and all leading and trailing dots are removed. This step avoids constructing a module name with illegal characters.

As an example, the JAR file name `ying-yang.jar` of an automatic module will derive the module name `ying.yang`, as will the JAR file name `ying-yang-1.2.3-ERC.jar`. File names for plain JARs acting as automatic modules should be chosen with care so as to avoid name clashes when such JARs do not specify a unique module name in their manifest file.

Explicit Modules

As the name implies, an explicit module is described by its module descriptor that explicitly specifies its name and the modules it reads (i.e., its dependencies), including any packages it exports or opens, or any services it provides or uses. A modular JAR is treated as an explicit module when specified on the module path.

Access to types in an explicit module is determined by the accessibility rules for types in packages *exported* or *opened* by the explicit module.

An explicit module *cannot* access code in the unnamed module as it cannot refer to the unnamed module in its module declaration.

Named Modules

Explicit modules and automatic modules are characterized as *named modules*. In the case of the explicit module, the name is specified in the module declaration. For the automatic modules, the name is either in the manifest file of its JAR or generated from its file name. A plain JAR or a modular JAR is treated as a named module when specified on the module path. Comparison of modules is shown in Table 19.5.

Table 19.5 *Module Comparison*

	Named modules		
	Explicit module	**Automatic module**	**Unnamed module**
JAR	Modular JAR	Plain JAR	Plain JAR or modular JAR
Path	On module path	On module path	On class path
Name	Specified in module descriptor	Specified in manifest file or from file name	Not applicable
Module descriptor	Mandatory	Not applicable	Not applicable
Reads	Only named modules that are required	All modules	All modules
Exports/Opens	Only exported/open packages	All packages	All packages

19.12 Migrating to Modules

The module system allows incremental migration of non-modular code to modules. Knowledge of dependencies between types in the plain JARs is crucial to drive this migration. This is probably easier with an application whose code one has control over than it is with libraries and frameworks from third parties. Regardless, having a better handle on dependencies makes the process easier. The `jar` tool (p. 1211) is useful to obtain information about the contents of a JAR and the `jdeps` tool (p. 1214) is a great tool to discover dependencies in the code.

The scenarios presented below are very much idealized, and deal with code in plain and modular JARs. The basic idea is to move code from the class path to the module path, either converting plain JARs to explicit modules directly or via automatic modules.

Bottom-Up Strategy for Code Migration

If all *direct* dependencies of a plain JAR are known to be modules, the plain JAR can be directly converted to an explicit module by declaring their dependencies and exports in a module declaration. This idea is embodied in the following algorithm, based on the graph of dependencies between the plain JARs (see Figure 19.16).

Figure 19.16 *Bottom-Up Strategy for Code Migration*

(1) Place all plain JARs on the *class path* such that they are in the unnamed module.

(2) Choose *leaf* JARs from the class path—that is, JARs that do not have any dependencies. Turn them into explicit modules by adding appropriate `exports` directives in their module descriptors and move them from the class path to the module path.

(3) From the class path, choose plain JARs from the next higher level, that have direct module dependencies. Turn them into explicit modules. Their module declarations must specify `requires` directives on their dependencies and appropriate `exports` directives for packages they export. Move the newly created explicit modules from the class path to the *module path.*

(4) Repeat step (3) until the unnamed module is empty.

There are variations on the bottom-up strategy, but the gist is represented by the algorithm above. Note that when applying the bottom-up strategy for migration, the explicit module on the module path never accesses code in any plain JAR on the class path.

Figure 19.16 shows the bottom-up strategy applied to the plain

JARs: `main.jar`, `controller.jar`, `view.jar`, and `model.jar`, that are converted to modular JARs—that is, explicit modules—as they are moved from the class path to the module path.

Top-Down Strategy for Code Migration

Migrating to modular code via *automatic* modules is convenient when it will take too long to wait for all dependencies to be available. The algorithm below migrates the code into explicit modules via automatic modules, based on the graph of known dependencies between the plain JARs (see Figure 19.17).

Figure 19.17 *Top-Down Strategy for Code Migration*

(1) Place all plain JARs on the module path. All plain JARs are now automatic modules.

(2) Choose *root* JARs that are automatic modules. A root JAR is one that no other JAR depends on. Turn these automatic modules into explicit modules by adding module declarations. Add `requires` directives using names of the automatic modules that a root JAR depends on.

(3) Choose automatic modules at the next lower level that are directly required by their parent modular JARs. Turn these automatic modules into explicit modules. Add `requires` directives using names of the automatic modules that these JARs depend on. Add appropriate `exports` directives for any packages exported.

(4) Repeat step (3) until all automatic modules have been converted to explicit modules.

Again, there are variations to the top-down strategy, but the algorithm embodies the main idea of this strategy. Note that in real life, both strategies will most likely be applied to complete the migration.

Figure 19.17 shows the top-down strategy applied to the plain JARs: `main.jar`, `controller.jar`, `view.jar`, and `model.jar`, which are first placed on the module path as automatic modules and then are converted to modular JARs—that is, explicit modules.

19.13 Exploring Modules

This section provides an introduction to using the JDK tools to discover, explore, and analyze modular applications.

Given the JARs `main.jar`, `control.jar`, `view.jar`, and `model.jar` for the `adviceApp` application in the `mlib` directory in Figure 19.14(c), p. 1188, we will use JDK tools to explore these archives for miscellaneous information about this application.

Listing the Contents of a JAR

We can list the contents of a JAR using the `--list` option (*short form*: `-t`) of the `jar` tool. The JAR file is specified with the `--file` option (*short form*: `-f`). The contents of `main.jar` are printed by the following `jar` command:

Click here to view code image

```
>jar --list --file mlib/main.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/
com/passion/
com/passion/main/
com/passion/main/Main.class
```

From the output on the terminal, we can see that there is a `META-INF` directory listed, in addition to the module descriptor file (`module-info.class`), the directory hierarchy of the package `com.passion.main`, and the file `Main.class` contained in it.

The `META-INF` directory is used internally to record any metadata in the JAR archive. In the listing above, the `META-INF` directory contains the file `MANIFEST.MF`. This manifest file is used to store any necessary information (specified as attribute-value pairs)—for example, the `Main-Class` attribute to designate the entry point of the application.

Extracting the Contents of a JAR

The `jar` tool can be used to extract the contents of a JAR into a desired directory. We illustrate the extract operation on `main.jar`. The following commands create the `main-extracted` directory under the current directory and change the working directory to it, respectively.

```
>mkdir main-extracted
>cd main-extracted
```

The `jar` command below with the `--extract` option (*short form*: `-x`) will extract the contents of `main.jar` into the `main-extracted` directory, which is now the current working directory. The structure of the `main-extracted` directory is also shown below after the extraction (printed using the `tree` command), showing the exploded module structure and the `META-INF` directory from `main.jar`.

Click here to view code image

```
>jar --extract --file ../mlib/main.jar
>tree -F --dirsfirst --noreport main-extracted
main-extracted
    META-INF/
        MANIFEST.MF
    com/
        passion/
            main/
                Main.class
    module-info.class
```

The contents of the `MANIFEST.MF` file, listed below using the `more` command, show that the entry point of the application is given by the value of `Main-Class`—that is, the class `com.passion.main.Main`.

Click here to view code image

```
>more main-extracted/META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 17.0.2 (Oracle Corporation)
Main-Class: com.passion.main.Main
```

Listing All Observable Modules

The `java` command with the `--list-modules` option (*no short form*) lists the *system modules* that are installed in the JDK, and then exits. These modules are available to every application. This lengthy list gives an idea of how the JDK has been modularized. The `java` command lists each module with its version; in this case, it is Java 17.0.2. Module names starting with the prefix `java` implement the Java SE Language Specification, whereas those starting with `jdk` are JDK specific. The reader will no doubt recognize some names, specially `java.base`.

The system modules are found in the `jmods` directory under the installation directory of the JDK. These are *JMOD files* having a module name with the extension `".jmod"`. JMOD files have a special non-executable format that allows native binary libraries and other configuration files to be packaged with bytecode artifacts that can then be linked to create runtime images with the `jlink` tool (p. 1222).

```
>java --list-modules
java.base@17.0.2
...
java.se@17.0.2
...
jdk.compiler@17.0.2
...
jdk.jartool@17.0.2
...
```

Specifying a module path in the `java` command below not only lists the system modules, but also the modular JARs found in the specified module path—in other words, all *observable modules*. In the `java` command below, the absolute path of all JARs found in the `mlib` directory is listed last in the output.

Click here to view code image

```
>java --module-path mlib --list-modules
java.base@17.0.2
...
jdk.javadoc@17.0.2
...
jdk.jdeps@17.0.2
...
controller file:.../adviceApp/mlib/controller.jar
main file:.../adviceApp/mlib/main.jar
model file:.../adviceApp/mlib/model.jar
view file:.../adviceApp/mlib/view.jar
```

Describing the Module Descriptor of a JAR

Both the `java` tool and the `jar` tool have the `--describe-module` option (*short form*: `-d`) to show the information contained in the module descriptor of a JAR. Both commands are used respectively below.

Click here to view code image

```
>java --module-path mlib --describe-module main
main file:.../adviceApp/mlib/main.jar
requires java.base mandated
requires controller
contains com.passion.main

>jar --file mlib/main.jar --describe-module
main jar:file:.../adviceApp/mlib/main.jar/!module-info.class
requires controller
requires java.base mandated
contains com.passion.main
main-class com.passion.main.Main
```

Note that in the `java` command, the `--describe-module` option requires a module name, whereas that is not the case in the `jar` command, where the `--file` option specifies the modular JAR.

Since the module descriptor is a Java bytecode class file, it must be disassembled to display its information. In both cases, first the module name is printed, followed by the path of the JAR. In the case of the `jar` command, the name `module-info.class` is appended to the path of the JAR. In both cases, the names of modules required (`java.base` and `controller`) are reported. The `main` module also *contains* an *internal* package (`com.passion.main`). Not surprisingly, the `java.base` module is mandated. However, only the `jar` command reports that the main-class is `com.passion.main.Main`. The `jar` command is useful to find the entry point of an application.

The `--describe-module` option (*short form*: `-d`) can also be used on a system module. The following `java` command describes the module descriptor of the `java.base` system module. The command lists all modules the `java.base` module *exports*, *uses*, *exports qualified*, and *contains* (p. 1177). As can be expected from its status as a mandated module for all other modules, it does *not require* any module.

Click here to view code image

```
>java --describe-module java.base
java.base@17.0.2
exports java.io
exports java.lang
...
uses java.util.spi.CurrencyNameProvider
uses java.util.spi.TimeZoneNameProvider
...
qualified exports jdk.internal to jdk.jfr
qualified exports sun.net.www to java.desktop java.net.http jdk.jartool
...
contains com.sun.crypto.provider
contains com.sun.java.util.jar.pack
...
```

Viewing Dependencies

The *Java Dependency Analysis tool*, `jdeps`, is a versatile command-line tool for static analysis of dependencies between Java artifacts like class files and JARs. It can analyze dependencies at all levels: module, package, and class. It allows the results to be filtered and aggregated in various ways, even generating various graphs to illustrate its findings. It is an indispensable tool when migrating non-modular code to make use of the module system.

In this section, we confine our discussion to modular code—that is, either an exploded module directory with the compiled module code (e.g., `mods/main`) or a modular JAR (e.g., `mlib/main.jar`).

In this section, the results from some of the `jdeps` commands have been edited to fit the width of the page without any loss of information, or elided to shorten repetitious outputs.

Viewing Package-Level Dependencies

The `jdeps` command with no options, shown below, illustrates the *default behavior* of the tool. Line numbers have been added for illustrative purposes.

When presented with the root module directory `mods/model` (1), the default behavior of `jdeps` is to print the name of the module (2), the path to its location (3), its module descriptor (4), followed by its module dependencies (5), and lastly the *package-level* dependencies (6). The package-level dependency at (6) shows that the package `com.passion.model` in the `model` module depends on the `java.lang` package in the proverbial `java.base` module.

Click here to view code image

```
(1)  >jdeps mods/model
(2)  model
(3)  [file:.../adviceApp/mods/model/]
(4)     requires mandated java.base (@17.0.2)
(5)  model -> java.base
(6)     com.passion.model           -> java.lang           java.base
```

The following `jdeps` command if let loose on the `model.jar` archive will print the same information for the `model` module, barring the difference in the file location:

```
>jdeps mlib/model.jar
```

However, the following `jdeps` command with the `main.jar` archive gives an error:

Click here to view code image

```
>jdeps mlib/main.jar
Exception in thread "main" java.lang.module.FindException: Module controller
not found, required by main
       at java.base/...
```

Dependency analysis cannot be performed on the `main` module by the `jdeps` tool because the modules the `main` module depends on cannot be found. In the case of the `model` module, which does not depend on any other user-defined module, the dependency analysis can readily be performed.

The two options `--module-path` (*no short form*) and `--module` (*short form*: `-m`) in the `jdeps` command below unambiguously specify the location of other modular JARs and the module to analyze, respectively. The format of the output is the same as before, and can be verified easily.

Click here to view code image

```
>jdeps --module-path mlib --module main
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
   requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main     -> com.passion.controller     controller
   com.passion.main     -> java.lang                  java.base
```

However, if one wishes to analyze all modules that the specified module depends on recursively, the `--recursive` option (*short form*: `-R`) can be specified. The output will be in the same format as before, showing the package-level dependencies for each module. The output from the `jdeps` command for each module is elided below, but has the same format we have seen previously.

Click here to view code image

```
>jdeps --module-path mlib --module main --recursive
controller
...
main
...
model
...
view
...
```

Viewing Module Dependencies

If the default output from the `jdeps` command is overwhelming, it can be filtered. The `-summary` option (*short form*: `-s`) will only print the *module dependencies*, as shown by the `jdeps` command below. Only the module dependencies of the `main` module will be shown in the output.

Click here to view code image

```
>jdeps --module-path mlib --module main -summary
main -> controller
main -> java.base
```

If we use the `--recursive` option (*short form*: `-R`) with the `-summary` option (*short form*: `-s`), then the module dependencies of each module will be printed recursively, starting with the specified module.

Click here to view code image

```
>jdeps --module-path mlib --module main -summary --recursive                (1)
controller -> java.base
controller -> model
controller -> view
main -> controller
main -> java.base
model -> java.base
view -> java.base
view -> model
```

Finally, we illustrate the graph-generating capabilities of the `jdeps` tool. The `jdeps` command takes all JARs of the `adviceApp` application from the `mlib` directory and creates a module graph (options `-summary` and `--recursive`) in the `DOT` format (option `-dotoutput`) under the current directory. The `DOT` file `summary.dot` containing the graph will be created. Using the `dot` command, this graph can be converted to a `pdf` file (`summary.pdf`) as shown in Figure 19.18.

Figure 19.18 *Module Graph Using the* `jdeps` *Tool*

Click here to view code image

```
>jdeps -dotoutput . -summary --recursive  mlib/*
>dot -Tpdf summary.dot >summary.pdf
```

The module graph in Figure 19.18 shows the same module dependencies printed by the `jdeps` command above at (1). Comparing the module graph in Figure 19.18 with the one in Figure 19.8, we see that `jdeps` has added the *implicit dependency* of each module on the `java.base` module.

Viewing Class-Level Dependencies

It is possible to dive deeper into dependencies with the `jdeps` tool. The `-verbose` option (*short form*: `-v`) will elicit the *class dependencies* of the specified module. Instead of package dependencies to round off the output, class dependencies are listed. The last line in the output shows that the class `com.passion.main.Main` in the `main` module depends on the `java.lang.String` class in the `java.base` module.

Click here to view code image

```
>jdeps --module-path mlib --module main -verbose
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
   requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main.Main  -> com.passion.controller.AdviceController  controll
   com.passion.main.Main  -> java.lang.Object                         java.bas
   com.passion.main.Main  -> java.lang.String                         java.bas
```

If we use the `--recursive` option (*short form*: `-R`) with the `-verbose` option (*short form*: `-v`), then the class dependencies of each module will be printed recursively, starting with the specified module.

Click here to view code image

```
>jdeps --module-path mlib --module main -verbose --recursive
controller
...
main
...
model
...
view
...
```

19.14 Summary of Selected Operations with the JDK Tools

Table 19.6 provides a ready reference for commands to perform miscellaneous operations using the JDK tools introduced in this chapter. Short forms for the options are shown where appropriate. Use the indicated reference for each operation to learn more about it. The only way to become familiar with these tools is to use them.

Table 19.6 *Selected Operations with the JDK Tools*

Operation	Command

Compiling modular code (*p. 1186*)

Click here to view code image

```
javac --module-path modulepath -d directory sourceAndModuleInfoFiles
javac  -p           modulepath -d directory sourceAndModuleInfoFiles
javac --module-source-path modulepath -d directory \
      --module moduleName1,...
javac --module-source-path modulepath -d directory \
       -m moduleName1,...
```

Launching modular application (*p. 1189*)

Click here to view code image

```
java --module-path modulepath --module moduleName/qualifiedClassName
java  -p           modulepath  -m      moduleName/qualifiedClassName
java --module-path modulepath  -module moduleName
java  -p           modulepath  -m      moduleName
```

Creating and listing a modular JAR archive (p. 1189)

Click here to view code image

```
jar --verbose --create --file jarfile -C directory files
jar -vcf jarfile -C directory .
jar -vcf jarfile --main-class qualifiedMainClassName -C directory .
jar -vcfe jarfile qualifiedMainClassName -C directory .

jar --list --file jarfile
jar -tf jarfile
```

Listing available modules (p. 1212)

Click here to view code image

```
java --list-modules
java --module-path modulepath --list-modules
java  -p           modulepath --list-modules
```

Describing a module—that is, printing the module descriptor (p. 1213)

Click here to view code image

```
java --module-path modulepath --describe-module moduleName
java  -p           modulepath  -d                moduleName

jar --file jarFile --describe-module
jar  -f    jarFile  -d
```

Viewing package-level dependencies (p. 1214)

Click here to view code image

```
jdeps --module-path modulepath -m moduleName
jdeps --module-path modulepath -m moduleName --recursive
jdeps --module-path modulepath -m moduleName -R
```

Viewing module dependencies (p. 1216)

Click here to view code image

```
jdeps --module-path modulepath -m moduleName -summary
jdeps --module-path modulepath -m moduleName -summary --recursive
jdeps --module-path modulepath -m moduleName -s -R
```

Viewing class-level dependencies (p. 1216)

Click here to view code image

```
jdeps --module-path modulepath -m moduleName -verbose
jdeps --module-path modulepath -m moduleName -verbose --recursive
jdeps --module-path modulepath -m moduleName -v -R
```

Selected Options for the `javac` Tool

The Java language compiler, `javac`, compiles Java source code into Java bytecode. The general form of the `javac` command is:

```
javac [options] [sourcefiles]
```

Table 19.7 shows some selected options that can be used when compiling modules. The optional *sourcefiles* is an itemized list of source files, often omitted in favor of using module-related options.

Table 19.7 *Selected Options for the* `javac` *Tool*

Option	Description
`--module-source-path` `moduleSourcePath`	The `moduleSourcePath` specifies the *source directory* where the exploded modules with the *source code files* can be found.
`--module-path modulepath` `-p modulepath`	The `modulepath` specifies where the modules needed by the application can be found. This can be a root directory of the exploded modules with the *class files* or a root directory where the

	modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on a Unix-based platform and semicolon (`;`) on the Windows platform.
`--module` *`moduleName`* `-m` *`moduleName`*	Specifies the module(s) to be compiled. Can be a single module name or a comma-separated (`,`) list of module names. For each module specified, all modules it depends on are also compiled, according to the module dependencies.
`-d` *`classesDirectory`*	Specifies the *destination directory* for the class files. Mandatory when compiling modules. For classes in a package, their class files are put in a directory hierarchy that reflects the package name, with directories being created as needed. Without the `-d` option, the class files are put in the same directory as their respective source files. Specifying the directory path is platform dependent: slash (`/`) on Unix-based platforms and backslash (`\`) on Windows platforms being used when specifying the directory path.
`--add-modules` *`module`*`,...`	Specifies root modules to resolve in addition to the initial modules. These modules can be modular JAR files, JMOD files, or even exploded modules.

Selected Options for the `java` Tool

The `java` tool launches an application—that is, it creates an instance of the JVM in which to run the application. A typical command to launch a modular application is by specifying the location of its modules (*path*) and the entry point of the application (as *module* or *module/mainclass*):

Click here to view code image

```
java --module-path path --module module[/mainclass]
```

Table 19.8 shows some selected options that can be used for launching and exploring modular applications.

Table 19.8 *Selected Options for the* `java` *Tool*

Option	Description
`--module-path` *`modulepath`* `-p` *`modulepath`*	The *`modulepath`* specifies the location where the modules needed to run the application can be found. This can be a root directory for the exploded modules with the *class files* or a directory where the modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on a Unix-based platform and semicolon (`;`) on Windows platforms.
`--module` *`module`*`[/`*`mainclass`*`]`	When *`module/mainclass`* is specified, it explicitly states the

`-m module[/mainclass]`	*module name* and the *fully qualified name of the class* with the `main()` method, thereby overriding any other entry point in the application. Without the *mainclass*, the entry point of the application is given by the *module* which must necessarily contain the main-class.
`--add-modules module,...`	Specifies root modules to resolve in addition to the initial modules.
`--list-modules`	Only lists the observable modules, and does not launch the application. That is, it lists the modules that the JVM can use when the application is run.
`--describe-module moduleName` `-d moduleName`	Describes a specified module, in particular its module descriptor, but does not launch the application.
`--validate-modules`	Validates all modules on the module path to find conflicts and errors within modules.

Selected Options for the `jar` Tool

The `jar` tool is an archiving and compression tool that can be used to bundle Java artifacts and any other resources that comprise the application. The archive file names have the `.jar` extension. A typical command to create a modular JAR (*`jarfile`*) with an application entry point (*`qualifiedMainClassName`*), based on the contents of a specific directory (*`DIR`*), is shown below. Note the obligatory dot (`.`) at the end of the command.

Click here to view code image

```
jar --create --file jarfile --main-class qualifiedMainClassName -C DIR .
```

Table 19.9 gives an overview of some selected options that can be used for working with JARs.

Table 19.9 *Selected Options for the* `jar` *Tool*

Option	Description
`--create` *or* `-c`	Creates a new archive.
`--extract` *or* `-x`	Extracts specified or all files in the archive.
`--list` *or* `-t`	Lists the contents of the archive.
`--update` *or* `-u`	Updates an existing archive with specified files.
`--describe-module` *or* `-d`	Prints the module descriptor of the archive and the main-class, if one is specified in the manifest.
`--verbose` *or* `-v`	Prints extra information about the operation.

`--file jarfile`
`--file=jarfile`
`-f jarfile`
`-f=jarfile`
	Specifies the name of the archive.

`-C DIR files`	Changes to the specified directory and includes the contents of the

	specified *files* from this directory. If *files* is a dot (`.`), the contents under the specified directory *DIR* are included.
Click here to view code image `--main-class` *`qualifiedMainClassName`* `--main-class=`*`qualifiedMainClassName`* `-e` *`qualifiedMainClassName`* `-e=`*`qualifiedMainClassName`*	Specifies the entry point of the application.
`--manifest` *`TXTFILE`* `--manifest=`*`TXTFILE`* `-m` *`TXTFILE`* `-m=`*`TXTFILE`*	Reads the manifest information for the archive from the specified *TXTFILE* and incorporates it in the archive—for example, the value of the `Main-Class` attribute that specifies the entry point of the application.
`--module-path` *`modulepath`* `-p` *`modulepath`*	Specifies the location of the modules for recording hashes.

Selected Options for the `jdeps` Tool

The Java Class Dependency Analyzer, `jdeps`, is the tool of choice when working with modules, as it is module savvy and highly versatile. Among its extensive module analyzing capabilities, it can be used to explore dependencies at different levels: module level, package level, and class level.

Table 19.10 gives an overview of some selected options for the `jdeps` tool that can be used for analyzing modules.

Table 19.10 *Selected Options for the* `jdeps` *Tool*

Option	Description
`--module-path` *`modulepath`*	Specifies where to find the module JARs needed by the application. No short form, as `-p` is already reserved for `--package`.
`--module` *`moduleName`* `-m` *`moduleName`*	Specifies the root module for module dependency analysis.
`--summary` *or* `-s`	Presents only a summary of the module dependencies.
`--recursive` *or* `-R`	Forces `jdeps` to recursively iterate over the module dependencies. When used alone, also prints the package-level dependencies.
`-verbose` *or* `-v`	Also includes all class-level dependencies in the printout.
`-verbose:package`	Includes package-level dependencies in the printout, excluding, by default, dependencies within the same package.
`-verbose:class`	Includes class-level dependencies in the printout, excluding, by default, dependencies within the same JAR.

Selected Options for the `jlink` Tool

The `jlink` tool creates a runtime image of an application. A typical command to create a runtime image of an application requires the location of its modules (*path*), names of modules to include (*module_names*), and output directory to store the runtime image (*output_dir*):

Click here to view code image

```
jlink --module-path path --add-modules module_names --output output_dir
```

The runtime image can be executed by the `output_dir/java` command.

Selected options for the `jlink` tool are summarized in Table 19.11.

Table 19.11 *Selected Options for the* `jlink` *Tool*

Option	Description
`--module-path modulepath` `-p modulepath`	Specifies the location where the modules for the application can be found. This can be a root directory for the exploded modules with the *class files* or a directory where the modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on Unix-based platforms and semicolon (`;`) on Windows platforms.
`--add-modules module,...`	Specifies the modules to include in the generated runtime image. All application modules must be listed. Any standard or JDK modules needed will be automatically included.
`--output path`	Specifies the location of the generated runtime image.

Final Remarks on Options for the JDK Tools

It is worth taking a note of how the command options having the short form `-p`, `-m`, and `-d` are specified in different JDK tools. Table 19.12 gives an overview of which long form they represent in which tool and how they are specified.

Table 19.12 *Selected Common Shorthand Options for JDK Tools*

javac	java	jar	jdeps
`--module-path path` `-p path`	`--module-path path` `-p path`	`--module-path path` *(no short form as* `-p` *is reserved for* `--package`*)*	
`--module module` `-m module`	`--module module[/mainclass]` `-m module[/mainclass]`	`--manifest FILE` `-m FILE`	
`-d classesDirectory` *(no long form)*	`--describe-module module` `-d module`	*(no long form)*	

Review Questions

19.1 Given the following code:

Click here to view code image

```
module store {
  requires transitive product;
  exports store.frontend to ui;
  exports store.backend;
}
module product {
  exports product.data;
}
module ui {
  requires store;
}
module customer {
  requires ui;
  exports customer.data;
}
```

Which statement is true?

Select the one correct answer.

a. The code in module `ui` can access public types from packages `store.frontend`, `store.backend`, `customer.data`, and `product.data`.

b. The code in module `ui` can access public types from packages `store.frontend`, `store.backend`, and `product.data`.

c. The code in module `customer` can access public types from package `store.frontend`, `store.backend`, and `product.data`.

d. The code in module `customer` can access public types from package `prod-uct.data`.

e. The code in module `product` can access public types from package `customer.data`.

19.2 Which statement is true about the `requires` directive?

Select the one correct answer.

a. It allows access to any public types in the module specified in the `requires` directive.

b. It allows access to non-public types using reflection in the module specified in the `requires` directive.

c. It allows access to protected types in the module specified in the `requires` directive.

d. It allows access to all types except protected types in the module specified in the `requires` directive.

e. It allows access to public types in the exported packages of the module specified in the `requires` directive.

19.3 Place the following modules in order, based on the number of dependencies they have, starting from the module that has the most dependencies.

Select the one correct answer.

a. `java.base java.se java.logging`
b. `java.logging java.se java.base`
c. `java.base java.logging java.se`
d. `java.logging java.base java.se`
e. `java.se java.logging java.base`
f. `java.se java.base java.logging`

19.4 Given the following code:

Click here to view code image

```
module zoo {
  requires animals;
}
module animals {
  exports animals.primates;
}
```

Which type declarations from the module `animals` can be accessed by code in the module `zoo`?
Select the two correct answers.

a.
```
package animals.primates;
class Ape { }
```

b.
```
package animals.primates;
protected class Gorilla { }
```

c.
```
package animals.primates;
public interface Toolmaker { }
```

d.
```
package animals.primates.fossil;
public class JavaMan { }
```

e.
```
package animals;
public class Hybrid { }
```

f. Click here to view code image

```
package animals.primates;
public class Chimpanzee extends Ape { }
```

19.5 The code in the `music` module needs to access types from package `production.company` contained in the `production` module.

Which module definitions implement this requirement?
Select the two correct answers.

a.
```
module music {
  requires production;
}
```

b.
```
module music {
  requires production.company;
}
```

c.
```
module music {
  requires production.company.*;
}
```

d.
```
module production {
  exports production.company;
}
```

e.
```
module production {
  exports production.*;
}
```

f.
```
module production {
  exports production.company.*;
}
```

19.6 Which of the following statements are true? Select the two correct answers.

a. An automatic module is a plain JAR loaded from the class path.

b. Plain JARs loaded from the module path are included in the unnamed module.

c. An automatic module is a plain JAR loaded from the module path.

d. Plain JARs loaded from the class path are included in the unnamed module.

19.7 Given the following code:

Click here to view code image

```
module store {
  requires product;
  exports store.frontend;
}

module marketing {
  requires store;
  opens marketing.offers;
}

module product {
  exports product.data;
  exports product.pricing to marketing;
}
```

Which statement is true?

Select the one correct answer.

a. The code in module `store` can access types defined in package `product.data`, but only by reflection in package `marketing.offers`.

b. The code in module `marketing` can access types defined in packages `product.data` and `product.pricing`.

c. The code in module `marketing` can access types defined in packages `product.data` and `store.frontend`.

d. The code in module `store` can access types defined in packages `product.data` and `product.pricing`.

e. The code in module `product` can access types defined in package `store.front-end`, and those in package `marketing.offers` by reflection.

19.8 The code contained in module `music` needs to access types from the `production.recording` package through reflection. It also needs to access public types from the packages `production.mix` and `artist.recording`.

Which module definitions implement these requirements? Select the one correct answer.

a. Click here to view code image

```
module music {
  requires artist.recording;
  requires production.mix;
}
module production {
  opens production.mix;
  exports production.recording;
}
module artist {
  exports artist.recording;
}
```

b.

```
module music {
  requires production;
}
module production {
  requires transitive artist;

  opens production.recording;
  exports production.mix;
}
module artist {
  exports artist.recording;
}
```

c. Click here to view code image

```
module music {
  requires production;
}
module production {
  opens production.mix;

  exports recording;
}
module artist {
  requires transitive production;

  exports artist.recording;
}
```

d. Click here to view code image

```
module music {
  requires production.recording;
}
module production {
  opens music;

  exports production.mix;
}
module artist {
  requires transitive production.recording;

  exports artist.recording;
}
```

19.9 Given the following code:

Click here to view code image

```
module music {
  exports music.sound;
}

module brass {
  requires music;
  provides music.sound.Instrument with brass.sound.Trumpet;
}
```

What is the correct definition for module `player` that wants to utilize the `music.sound.Instrument` service?
Select the one correct answer.

a. Click here to view code image

```
module player {
  requires brass;
  uses music.sound.Instrument;
}
```

b. Click here to view code image

```
module player {
  requires music;
  uses music.sound.Instrument;
}
```

c.

```
module player {
  requires brass;
  uses brass.sound.Trumpet;
}
```

d.

```
module player {
  requires music;
  uses brass.sound.Trumpet;
}
```

19.10 Which of the following statements are true about the `jlink` tool and runtime images?

Select the two correct answers.

a. The `jlink` tool can create platform-independent runtime images.

b. The `jlink` tool can create platform-specific runtime images.

c. Runtime images only contain application code.

d. Runtime images are automatically updated when a new Java version is installed.

e. The runtime image of an application can be executed on a system that has no JVM installed.

19.11 Given the following code:

Click here to view code image

```
module music {
  requires artist;
  requires instrument;
  exports preferences.style to catalog;
}

module instrument {
  requires transitive music;
  opens instrument.data;
}

module artist {
  exports preferences.style;
}

module catalog {
}
```

What are the reasons why the module declarations will fail to compile?
Select the two correct answers.

a. There is a cyclic dependency between modules.

b. There is a split package deployment.

c. There is an `opens` directive in a non-open module.

d. There is a qualified export referring to a module that does not require this export.

19.12 Which statement is true?

Select the one correct answer.

a. A modular JAR is not backward compatible and thus cannot be used in the context of a class path.

b. A plain JAR is not forward compatible and thus cannot be used in the context of a module path.

c. The `--list-modules` option of the `java` command includes the unnamed module.

d. The `--list-modules` option of the `java` command does not include automatic modules.

e. By default, an automatic module is referred to by its JAR file name.

f. By default, the unnamed module is referred to by its JAR file name.

19.13 Which of the following statements are true? Select the two correct answers.

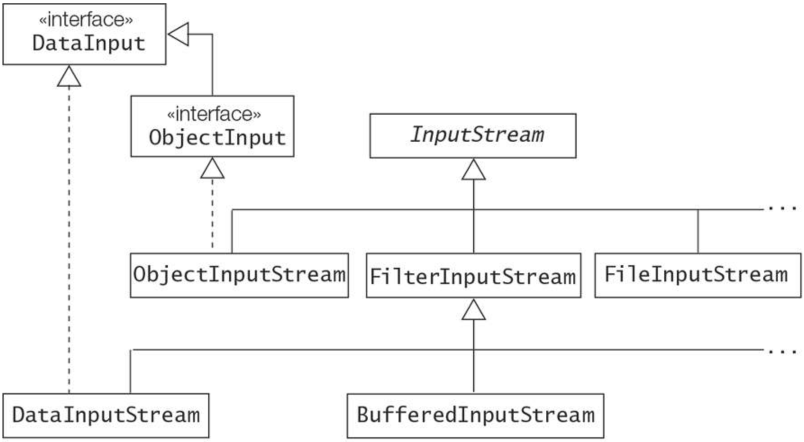
a. An automatic module implicitly requires all explicit modules, but not other automatic modules or the unnamed module.

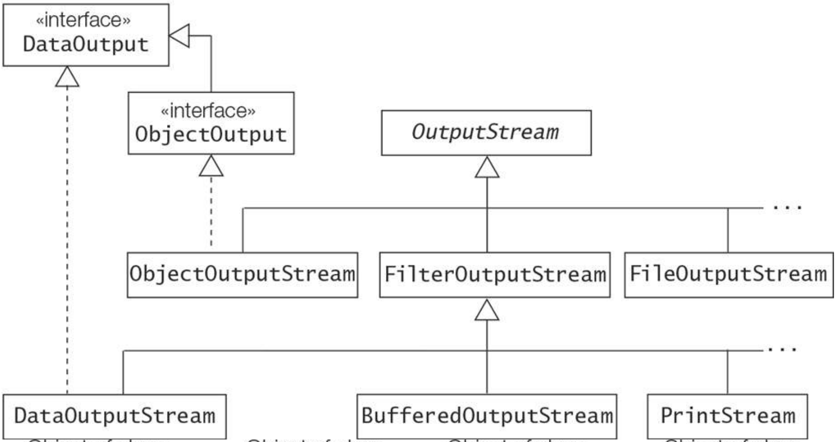
b. An automatic module implicitly requires all explicit modules and other automatic modules, but not the unnamed module.

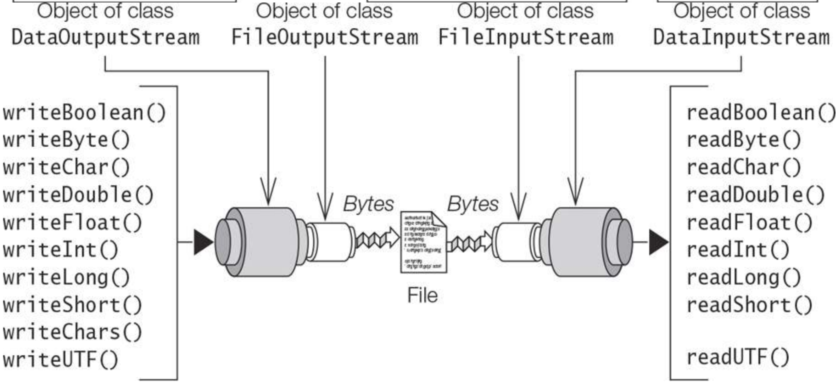
c. An automatic module implicitly requires all other modules, including explicit and automatic modules, and the unnamed module.

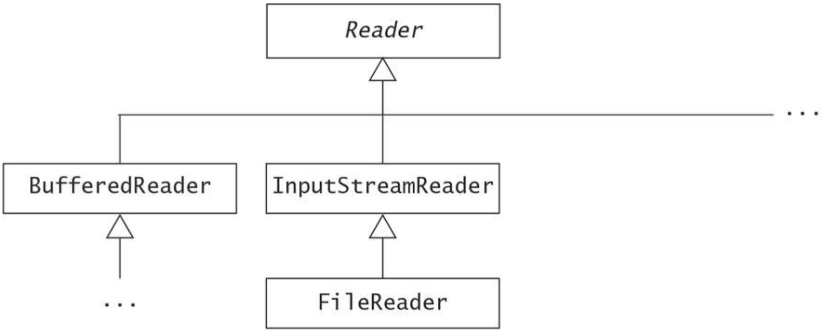
d. An explicit module must use the `requires` directive to access types in an automatic module.

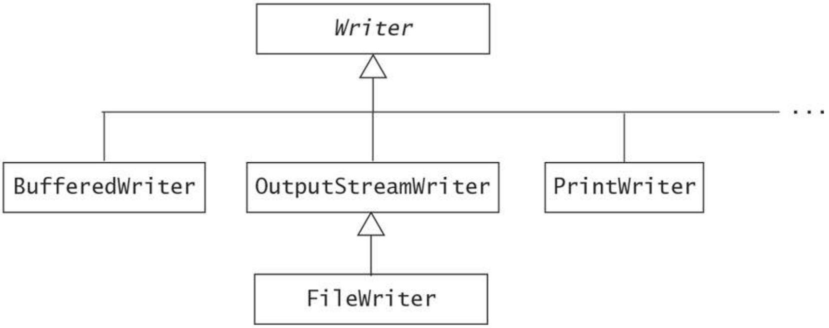
e. An explicit module must use the `requires` directive to access types in the unnamed module.

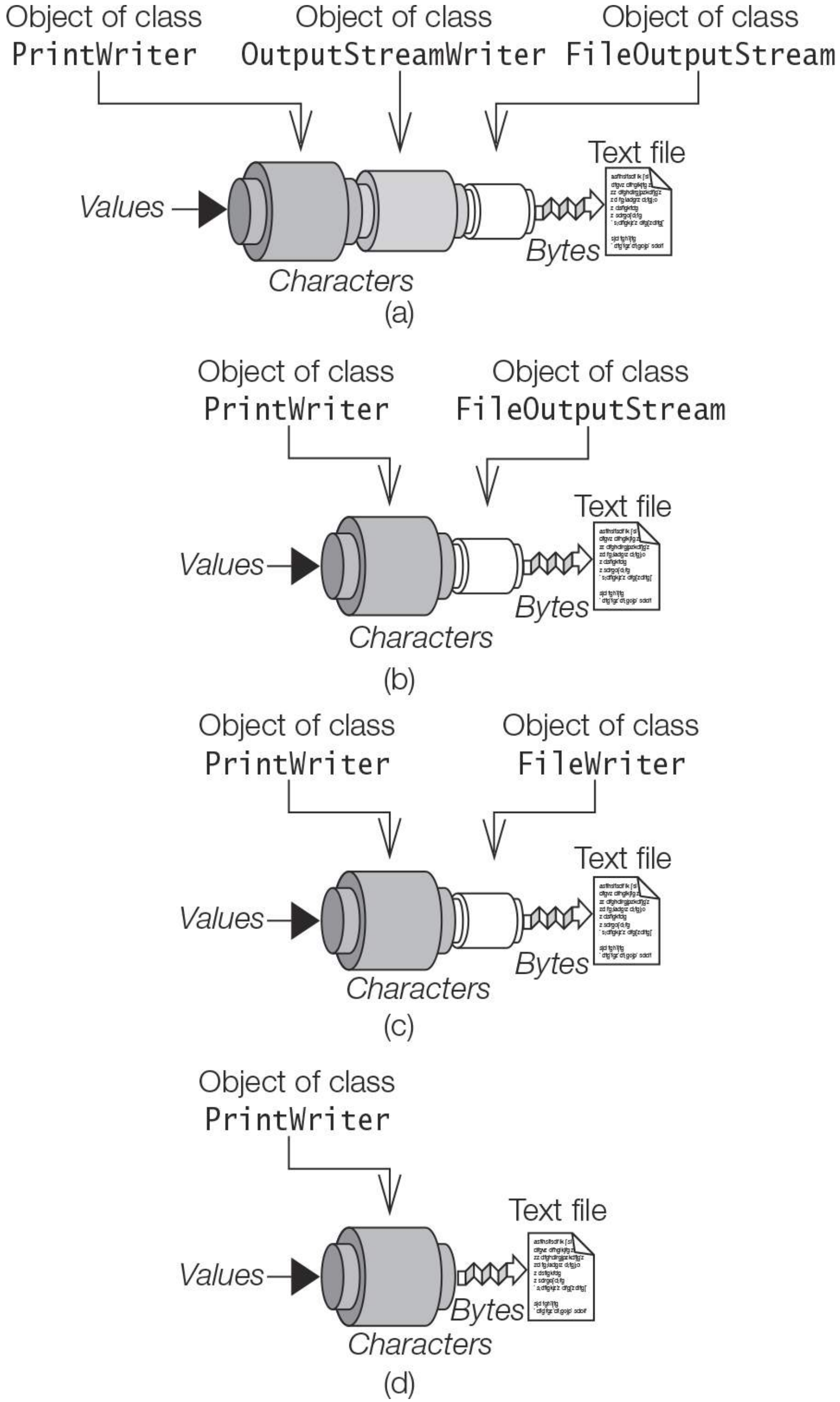

«interface»
DataInput
«interface»
ObjectInput
InputStream
ObjectInputStream
FilterInputStream
FileInputStream
DataInputStream
BufferedInputStream

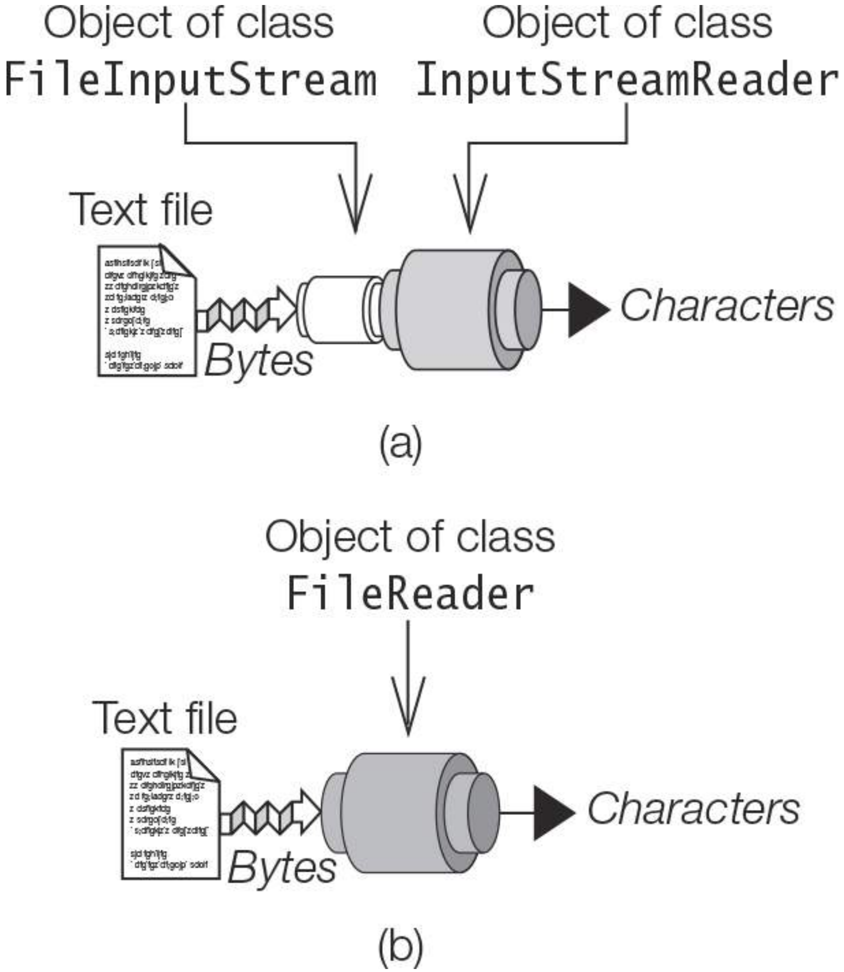

«interface»
DataOutput
«interface»
ObjectOutput
OutputStream
ObjectOutputStream
FilterOutputStream
FileOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

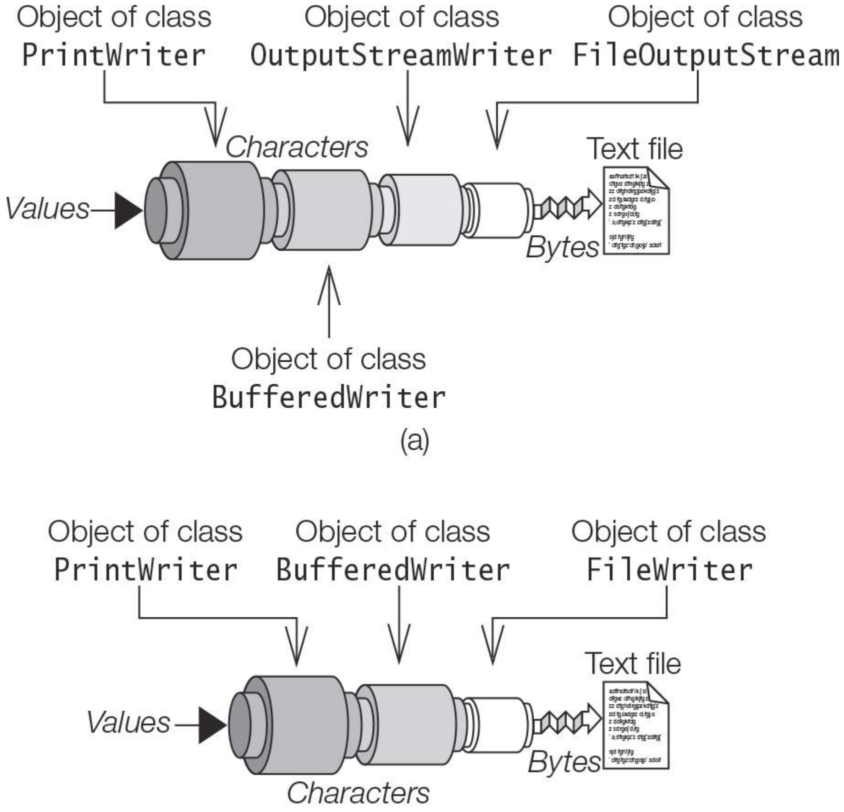

Object of class
DataOutputStream
Object of class
FileOutputStream
Object of class
FileInputStream
Object of class
DataInputStream
writeBoolean()
writeByte()
writeChar()
writeDouble()
writeFloat()
writeInt()
writeLong()
writeShort()
writeChars()
writeUTF()
Bytes
Bytes
File
readBoolean()
readByte()
readChar()
readDouble()
readFloat()
readInt()
readLong()
readShort()
readUTF()


Reader
BufferedReader
InputStreamReader
FileReader
...
...

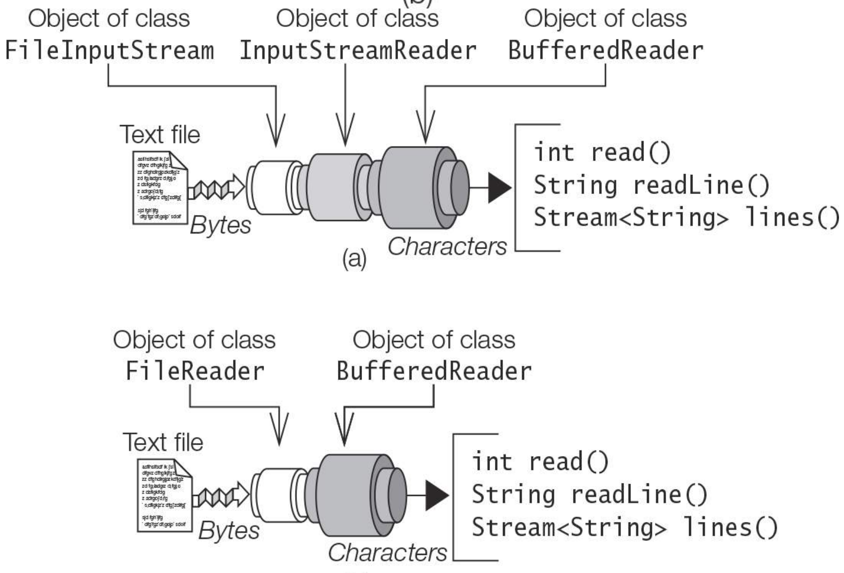

Writer
BufferedWriter
OutputStreamWriter
PrintWriter
FileWriter
...

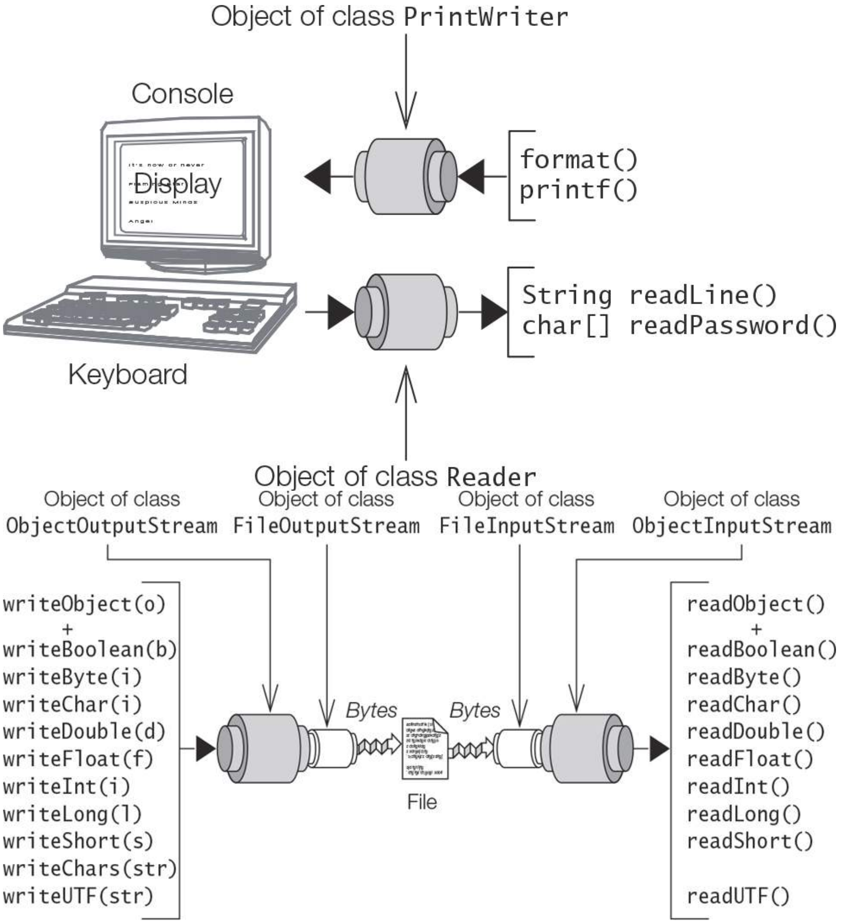

Object of class
PrintWriter
Object of class
OutputStreamWriter
Object of class
FileOutputStream
Text file
Values
Bytes
Characters
(a)
Object of class
PrintWriter
Object of class
FileOutputStream
Text file
Values
Bytes
Characters
(b)
Object of class
PrintWriter
Object of class
FileWriter
Text file
Values
Bytes
Characters
(c)
Object of class
PrintWriter
Text file
Values
Bytes
Characters
(d)


Object of class
FileInputStream
Object of class
InputStreamReader
Text file
Bytes
Characters
(a)
Object of class
FileReader
Text file
Bytes
Characters
(b)


Object of class PrintWriter
Object of class OutputStreamWriter
Object of class FileOutputStream
Characters
Text file
Values
Bytes
Object of class BufferedWriter
(a)
Object of class PrintWriter
Object of class BufferedWriter
Object of class FileWriter
Text file
Values
Bytes
Characters

(b)


Object of class FileInputStream
Object of class InputStreamReader
Object of class BufferedReader
Text file
Bytes
Characters
int read()
String readLine()
Stream<String> lines()
(a)
Object of class FileReader
Object of class BufferedReader
Text file
Bytes
Characters
int read()
String readLine()
Stream<String> lines()
(b)


Object of class PrintWriter
Console
Display
format()
printf()
String readLine()
char[] readPassword()
Keyboard
Object of class Reader
Object of class ObjectOutputStream
Object of class FileOutputStream
Object of class FileInputStream
Object of class ObjectInputStream
writeObject(o)
+
writeBoolean(b)
writeByte(i)
writeChar(i)
writeDouble(d)
writeFloat(f)
writeInt(i)
writeLong(l)
writeShort(s)
writeChars(str)
writeUTF(str)
Bytes
Bytes
File
readObject()
+
readBoolean()
readByte()
readChar()
readDouble()
readFloat()
readInt()
readLong()
readShort()
readUTF()

Java I/O: Part I

20

Chapter Topics

- Understanding the concept of I/O streams to read data from an input source and write data to an output destination
- Understanding the hierarchy of byte streams, as represented by the `InputStream` and `OutputStream` abstract classes, to handle data as bytes
- Reading and writing bytes using file streams, as represented by the `FileInputStream` and `FileOutputStream` classes
- Reading and writing binary files using the `DataInputStream` and `DataOutputStream` classes
- Understanding the hierarchy of character streams, as represented by the `Reader` and `Writer` abstract classes
- Using character encodings, including Unicode and UTF-8, with the `InputStreamReader` and `OutputStreamWriter` classes
- Reading and writing text files using the `FileReader` and `PrintWriter` classes
- Using buffered character streams for efficient reading and writing of characters, as represented by the `BufferedReader` and `Buffered-Writer` classes
- Understanding the standard input, output, and error streams, as represented by the fields `System.in`, `System.out`, and `System.err`, respectively
- Using the `Console` class that allows character-based input and output from a console
- Performing object serialization/de-serialization—that is, writing and reading binary representation of objects, using the `ObjectOutputStream` and `ObjectInputStream` classes
- Applying selective serialization, exploiting customized serialization, using class versioning, and understanding how the inheritance relationship can affect serialization

Java SE 17 Developer Exam O	
[9.1] Read and write console and file data using I/O Streams	*§20.1, p. 1233* *to* *§20.4, p. 1256*
[9.2] Serialize and de-serialize Java objects	*§20.5, p. 1261*
Java SE 11 Developer Exam O	

9.1 Read and write console and file data using I/O Streams	*§20.1, p. 1233* *to* *§20.4, p. 1256*
9.2 Implement serialization and deserialization techniques on Java objects	*§20.5, p. 1261*

The `java.io` package provides an extensive library of classes for dealing with input and output of data. Although data can be read from various sources and written to various destinations, the emphasis in this chapter is on using the standard I/O API provided by the `java.io` package for reading and writing various kinds of data to *files*: bytes, characters, binary files, text files, buffered I/O, and object serialization.

The `java.io` package also provides a general interface to interact with the file system of the host platform, but we will use the newer and enhanced I/O API in the `java.nio.file` package that provides access to files, file attributes, and file systems (Chapter 21, p. 1285).

In this chapter, we assume a basic understanding of a *hierarchical file system* in which files and directories are referenced by a *string* that specifies the *path* of an entity in the file system. The idiosyncrasies of file systems on different platforms are explored in Chapter 21, p. 1285.

It is important to note that the term *stream* in this chapter refers to *I/O streams*. Streams that provide functional-style operations on sequences of elements are discussed in Chapter 16, p. 879.

20.1 Input and Output

Java provides *I/O streams* as a general mechanism for dealing with data input and output. I/O streams implement *sequential processing* of data. An *input stream* allows an application to read a sequence of data, and an *output stream* allows an application to write a sequence of data. An *input stream* acts as a *source* of data, and an *output stream* acts as a *destination* of data. The following entities can act as both input and output streams:

- A file—which is the focus in this chapter
- An array of bytes or characters
- A network connection

There are two categories of I/O streams:

- *Byte streams* that process *bytes* as a unit of data
- *Character streams* that process *characters* as a unit of data

A *low-level I/O stream* operates directly on the data source (e.g., a file or an array of bytes), and processes the data primarily as *bytes*.

A *high-level I/O stream* is *chained* to an underlying stream, and provides additional capabilities for processing data managed by the underlying stream—for example, processing bytes from the underlying stream as Java primitive values or objects. In other words, a high-level I/O stream acts as a *wrapper* for the underlying stream.

In the rest of this chapter we primarily explore how to use I/O streams of the standard I/O API provided by the `java.io` package to read and write various kinds of data that is stored in files.

20.2 Byte Streams: Input Streams and Output Streams

The abstract classes `InputStream` and `OutputStream` in the `java.io` package are the root of the inheritance hierarchies for handling the reading and writing of data as *bytes* (Figure 20.1). Their subclasses, implementing different kinds of input and output (I/O) streams, override the following methods from the `InputStream` and `OutputStream` classes to customize the reading and writing of bytes, respectively:

Figure 20.1 *Partial Byte Stream Inheritance Hierarchies in the* `java.io` *Package*

From the InputStream abstract class:

Click here to view code image

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

Note that the first `read()` method reads a *byte*, but returns an `int` value. The byte read resides in the eight least significant bits of the `int` value, while the remaining bits in the `int` value are zeroed out. The `read()` methods return the value –1 when the end of the input stream is reached.

Click here to view code image

> ```
> long transferTo(OutputStream out) throws IOException
> ```
>
> Reads bytes from this input stream and writes to the specified output stream in the order they are read. The I/O streams are *not closed* after the operation (see below).

From the `OutputStream` abstract class:

> Click here to view code image
>
> ```
> void write(int b) throws IOException
> void write(byte[] b) throws IOException
> void write(byte[] b, int off, int len) throws IOException
> ```
>
> The first `write()` method takes an `int` as an argument, but truncates it to the eight least significant bits before writing it out as a byte to the output stream.

> Click here to view code image
>
> ```
> void close() throws IOException          Both InputStream and OutputStream
> void flush() throws IOException          Only for OutputStream
> ```
>
> A I/O stream should be *closed* when no longer needed, to free system resources. Closing an output stream automatically *flushes* the output stream, meaning that any data in its internal buffer is written out.
>
> Since byte streams also implement the `AutoCloseable` interface, they can be declared in a `try`-with-resources statement (§7.7, p. 407) that will ensure they are properly closed after use at runtime.
>
> An output stream can also be manually flushed by calling the second method.

Read and write operations on streams are *blocking* operations—that is, a call to a read or write method does not return before a byte has been read or written.

Many methods in the classes contained in the `java.io` package throw the checked `IOException`. A calling method must therefore either catch the exception explicitly, or specify it in a `throws` clause.

Table 20.1 and Table 20.2 give an overview of selected byte streams. Usually an output stream has a corresponding input stream of the same type.

Table 20.1 *Selected Input Streams*

`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterInputStream`	The superclass of all *input filter streams*. An input filter stream must be chained to an underlying input stream.
`DataInputStream`	A filter stream that allows the *binary representation of Java primitive values* to be read from an underlying input stream. The underlying input stream must be specified.
`ObjectInputStream`	A filter stream that allows *binary representations of Java objects and Java primitive values* to be read from a specified input stream.

Table 20.2 *Selected Output Streams*

`FileOutputStream`	Data is written as *bytes* to a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterOutputStream`	The superclass of all *output filter streams*. An output filter stream must be chained to an underlying output stream.
`PrintStream`	A filter output stream that converts a *text representation of Java objects and Java primitive values* to bytes before writing them to an underlying output stream, which must be specified. This is the type of `System.out` and `System.err` (p. 1255). However, the `PrintWriter` class is recommended when writing characters rather than bytes (p. 1247).
`DataOutputStream`	A filter stream that allows the *binary representation of Java primitive values* to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	A filter stream that allows the *binary representation of Java objects and Java primitive values* to be written to a specified underlying output stream.

File Streams

The subclasses `FileInputStream` and `FileOutputStream` represent low-level streams that define byte input and output streams that are connected to files. Data can only be read or written as a sequence of *bytes*. Such file streams are typically used for handling image data.

A `FileInputStream` for reading bytes can be created using the following constructor:

Click here to view code image

```
FileInputStream(String name) throws FileNotFoundException
```

The file designated by the file name is assigned to a new file

> input stream.
>
> If the file does not exist, a `FileNotFoundException` is thrown. If it exists, it is set to be read from the beginning. A `SecurityException` is thrown if the file does not have read access.

A `FileOutputStream` for writing bytes can be created using the following constructor:

> Click here to view code image
>
> ```
> FileOutputStream(String name) throws FileNotFoundException
> FileOutputStream(String name, boolean append) throws FileNotFoundException
> ```
>
> The file designated by the file name is assigned to a new file output stream.
>
> If the file does not exist, it is created. If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A `SecurityException` is thrown if the file does not have write access or it cannot be created. A `FileNotFoundException` is thrown if it is not possible to open the file for any other reasons.

The `FileInputStream` class provides an implementation for the `read()` methods in its superclass `InputStream`. Similarly, the `FileOutputStream` class provides an implementation for the `write()` methods in its superclass `OutputStream`.

Example 20.1 demonstrates using a buffer to read bytes from and write bytes to file streams. The input and the output file names are specified on the command line. The streams are created at (1) and (2).

The bytes are read into a buffer by the `read()` method that returns the number of bytes read. The same number of bytes from the buffer are written to the output file by the `write()` method, regardless of whether the buffer is full or not after every read operation.

The end of file is reached when the `read()` method returns the value `-1`. The code at (3a) using a buffer can be replaced by a call to the `transferTo()` method at (3b) to do the same operation. The streams are closed by the `try`-with-resources statement. Note that most of the code consists of a `try`-with-resources statement with `catch` clauses to handle the various exceptions.

Example 20.1 *Copying a File Using a Byte Buffer*

Click here to view code image

```
/* Copy a file using a byte buffer.
   Command syntax: java CopyFile <from_file> <to_file> */
import java.io.*;

class CopyFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileInputStream fromFile = new FileInputStream(args[0]);        // (1)
        FileOutputStream toFile = new FileOutputStream(args[1]))   {    // (2)

      // Copy bytes using buffer:                                       // (3a)
      byte[] buffer = new byte[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer bytes:
//    fromFile.transferTo(toFile);                                      // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyFile <from_file> <to_file>");
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

I/O Filter Streams

An *I/O filter stream* is a high-level I/O stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter stream. The `FilterInputStream` and `FilterOutputStream` classes, together with their subclasses, define input and output filter streams. The subclasses `BufferedInputStream` and `BufferedOutputStream` implement filter streams that buffer input from and output to the underlying stream, respectively. The subclasses `DataInputStream` and `DataOutputStream` implement filter streams that allow binary representation of Java primitive values to be read and written, respectively, from and to an underlying stream.

Reading and Writing Binary Values

The `java.io` package contains the two interfaces `DataInput` and `DataOutput`, which streams can implement to allow reading and writing of *binary representation of Java primitive values* (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`). The methods for writing binary representations of Java primitive values are named `write`*X*, where *X* is any Java primitive data type. The methods for reading binary representations of Java primitive values are similarly named `read`*X*. Table 20.3 gives an overview of the `read`*X*`()` and `write`*X*`()` methods found in these two interfaces. A file containing *binary values* (i.e., binary representation of Java primitive values) is usually called a *binary file*.

Table 20.3 *The* `DataInput` *and* `DataOutput` *Interfaces*

Type	Methods in the `DataInput` interface	Methods in the `DataOutput` interface
`boolean`	`readBoolean()`	`writeBoolean(boolean b)`
`char`	`readChar()`	`writeChar(int c)`
`byte`	`readByte()`	`writeByte(int b)`
`short`	`readShort()`	`writeShort(int s)`
`int`	`readInt()`	`writeInt(int i)`
`long`	`readLong()`	`writeLong(long l)`
`float`	`readFloat()`	`writeFloat(float f)`
`double`	`readDouble()`	`writeDouble(double d)`
`String`	[illegible]	`writeChars(String str)`
`String`	`readUTF()`	`writeUTF(String str)`

Note the methods provided for reading and writing strings. However, the recommended practice for reading and writing characters is to use *character streams*, called *readers* and *writers*, which are discussed in §20.3.

The filter streams `DataOutputStream` and `DataInputStream` implement the `DataOutput` and `DataInput` interfaces, respectively, and can be used to read and write binary representation of Java primitive values from and to an underlying stream. Both the `write`*X*`()` and `read`*X*`()` methods throw an `IOException` in the event of an I/O error. In particular, the `read`*X*`()` methods throw an `EOFException` (a subclass of `IOException`) if the input stream does not contain the correct number of bytes to read. Bytes can also be skipped from a `DataInput` stream, using the `skipBytes(int n)` method which skips `n` bytes.

> Click here to view code image
>
> ```
> DataInputStream(InputStream in)
> DataOutputStream(OutputStream out)
> ```
>
> These constructors can be used to set up filter streams from an underlying stream for reading and writing Java primitive values, respectively.

Writing Binary Values to a File

To write the binary representation of Java primitive values to a *binary file*, the following procedure can be used, which is also depicted in Figure 20.2.

Figure 20.2 *Stream Chaining for Reading and Writing Binary Values to a File*

- Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
- Create a `FileOutputStream`:

  Click here to view code image

  ```
  FileOutputStream outputFile = new FileOutputStream("primitives.data")
  ```

- Create a `DataOutputStream` which is chained to the `FileOutputStream`:

  Click here to view code image

  ```
  DataOutputStream outputStream = new DataOutputStream(outputFile);
  ```

- Write Java primitive values using relevant `writeX()` methods:

  Note that in the case of `char`, `byte`, and `short` data types, the `int` argument to the `writeX()` method is converted to the corresponding type, before it is written (see Table 20.3).

See also the numbered lines in Example 20.2 corresponding to the steps above.

Reading Binary Values from a File

To read the binary representation of Java primitive values from a *binary file*, the following procedure can be used, which is also depicted in Figure 20.2.

- Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
- Create a `FileInputStream`:

  Click here to view code image

  ```
  FileInputStream inputFile = new FileInputStream("primitives.data");
  ```

- Create a `DataInputStream` which is chained to the `FileInputStream`:

  Click here to view code image

  ```
  DataInputStream inputStream = new DataInputStream(inputFile);
  ```

- Read the (exact number of) Java primitive values in the *same order* they were written out to the file, using relevant `read`*X*`()` methods. Not doing so will unleash the wrath of the `IOException`.

See also the numbered lines in Example 20.2 corresponding to the steps above. Example 20.2 uses both procedures described above: first to write and then to read some Java primitive values to and from a file. It also checks to see if the end of the stream has been reached, signaled by an `EOFException`. The values are also written to the standard output stream.

Example 20.2 *Reading and Writing Binary Values*

Click here to view code image

```
import java.io.*;

public class BinaryValuesIO {
  public static void main(String[] args) throws IOException {

    // Write binary values to a file:
    try(                                                                  //
        // Create a FileOutputStream.
        FileOutputStream outputFile = new FileOutputStream("primitives.data");
        // Create a DataOutputStream which is chained to the FileOutputStream.
        DataOutputStream outputStream = new DataOutputStream(outputFile)) {

      // Write Java primitive values in binary representation:
      outputStream.writeBoolean(true);
      outputStream.writeChar('A');              // int written as Unicode cha
      outputStream.writeByte(Byte.MAX_VALUE);   // int written as 8-bits byte
      outputStream.writeShort(Short.MIN_VALUE); // int written as 16-bits sho
      outputStream.writeInt(Integer.MAX_VALUE);
      outputStream.writeLong(Long.MIN_VALUE);
      outputStream.writeFloat(Float.MAX_VALUE);
      outputStream.writeDouble(Math.PI);
    }

    // Read binary values from a file:
    try (                                                                 //
        // Create a FileInputStream.
        FileInputStream inputFile = new FileInputStream("primitives.data");

        // Create a DataInputStream which is chained to the FileInputStream.
        DataInputStream inputStream = new DataInputStream(inputFile)) {
```

```
      // Read the binary representation of Java primitive values
      // in the same order they were written out:
      System.out.println(inputStream.readBoolean());
      System.out.println(inputStream.readChar());
      System.out.println(inputStream.readByte());
      System.out.println(inputStream.readShort());
      System.out.println(inputStream.readInt());
      System.out.println(inputStream.readLong());
      System.out.println(inputStream.readFloat());
      System.out.println(inputStream.readDouble());

      // Check for end of stream:
      int value = inputStream.readByte();
      System.out.println("More input: " + value);
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found.");
    } catch (EOFException eof) {
      System.out.println("End of input stream.");
    }
  }
}
```

Output from the program:

```
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
End of input stream.
```

20.3 Character Streams: Readers and Writers

A *character encoding* is a scheme for representing characters. Java programs represent values of the `char` type internally in the 16-bit Unicode character encoding, but the host platform might use another character encoding to represent and store characters externally. For example, the ASCII (American Standard Code for Information Interchange) character encoding is widely used to represent characters on many platforms. However, it is only one small subset of the Unicode standard.

The abstract classes `Reader` and `Writer` are the roots of the inheritance hierarchies for streams that read and write *Unicode characters* using a specific character encoding (Figure 20.3). A *reader* is an input character stream that implements the `Readable` interface and reads a sequence of Unicode characters, and a *writer* is an output character stream that implements the `Writer` interface and writes a sequence of Unicode characters. Character encodings (usually called *charsets*) are used by readers and writers to convert between external bytes and internal Unicode characters. The same character encoding that was used to write the characters must be used to read those characters. The `java.nio.charset.Charset` class represents charsets. Kindly refer to the `Charset` class API documentation for more details.

Figure 20.3 *Selected Character Streams in the* `java.io` *Package*

> Click here to view code image
>
> ```
> static Charset forName(String charsetName)
> ```
>
> Returns a charset object for the named charset. Selected common charset names are `"UTF-8"`, `"UTF-16"`, `"US-ASCII"`, and `"ISO-8859-1"`.
>
> Click here to view code image
>
> ```
> static Charset defaultCharset()
> ```
>
> Returns the default charset of this Java virtual machine.

Table 20.4 and Table 20.5 give an overview of some selected character streams found in the `java.io` package.

Table 20.4 *Selected Readers*

Reader	**Description**
`BufferedReader`	A reader is a high-level input stream that buffers the characters read from an underlying stream. The underlying stream must be specified and an optional buffer size can be given.
`InputStreamReader`	Characters are read from a byte input stream which must be specified. The default character encoding is used if no character encoding is explicitly specified in the constructor. This class provides the bridge from byte streams to character streams.
`FileReader`	Characters are read from a file, using the default character encoding, unless an encoding is explicitly specified in the constructor. The file can be specified by a `String` file name. It automatically creates a `FileInputStream` that is associated with the file.

Table 20.5 *Selected Writers*

Writers	**Description**
`BufferedWriter`	A writer is a high-level output stream that buffers the characters before writing them to an underlying stream. The underlying stream must be specified, and an optional buffer size can be specified.
`OutputStreamWriter`	

	Characters are written to a byte output stream that must be specified. The default character encoding is used if no explicit character encoding is specified in the constructor. This class provides the bridge from character streams to byte streams.
`FileWriter`	Characters are written to a file, using the default character encoding, unless an encoding is explicitly specified in the constructor. The file can be specified by a `String` file name. It automatically creates a `FileOutputStream` that is associated with the file. A `boolean` parameter can be specified to indicate whether the file should be overwritten or appended with new content.
`PrintWriter`	A print writer is a high-level output stream that allows *text representation of Java objects and Java primitive values* to be written to an underlying output stream or writer. The underlying output stream or writer must be specified. An explicit encoding can be specified in the constructor, and also whether the print writer should do automatic line flushing.

Readers use the following methods for reading Unicode characters:

> Click here to view code image
>
> ```
> int read() throws IOException
> int read(char cbuf[]) throws IOException
> int read(char cbuf[], int off, int len) throws IOException
> ```
>
> Note that the `read()` methods read the character as an `int` in the range 0 to 65,535 (0x0000–0xFFFF).
>
> The first method returns the character as an `int` value. The last two methods store the characters in the specified array and return the number of characters read. The value `-1` is returned if the end of the stream has been reached.
>
> Click here to view code image
>
> ```
> long skip(long n) throws IOException
> ```
>
> A reader can skip over characters using the `skip()` method.
>
> Click here to view code image
>
> ```
> void close() throws IOException
> ```
>
> Like byte streams, a character stream should be closed when no longer needed in order to free system resources.
>
> Click here to view code image
>
> ```
> boolean ready() throws IOException
> ```
>
> When called, this method returns `true` if the next read

> operation is guaranteed not to block. Returning `false` does *not* guarantee that the next read operation will block.
>
> Click here to view code image
>
> ```
> long transferTo(Writer out) throws IOException
> ```
>
> Reads all characters from this reader and writes the characters to the specified writer in the order they are read. The I/O streams are not *closed* after the operation.

Writers use the following methods for writing Unicode characters:

> Click here to view code image
>
> ```
> void write(int c) throws IOException
> ```
>
> The `write()` method takes an `int` as an argument, but writes only the least significant 16 bits.
>
> Click here to view code image
>
> ```
> void write(char[] cbuf) throws IOException
> void write(String str) throws IOException
> void write(char[] cbuf, int off, int length) throws IOException
> void write(String str, int off, int length) throws IOException
> ```
>
> Write the characters from an array of characters or a string.
>
> Click here to view code image
>
> ```
> void close() throws IOException
> void flush() throws IOException
> ```
>
> Like byte streams, a character stream should be closed when no longer needed in order to free system resources. Closing a character output stream automatically *flushes* the stream. A character output stream can also be manually flushed.

Like byte streams, many methods of the character stream classes throw a checked `IOException` that a calling method must either catch explicitly or specify in a `throws` clause. They also implement the `AutoCloseable` interface, and can thus be declared in a `try`-with-resources statement (§7.7, p. 407) that will ensure they are automatically closed after use at runtime.

Analogous to Example 20.1 that demonstrates usage of a byte buffer for writing and reading bytes to and from file streams, Example 20.3 demonstrates using a character buffer for writing and reading characters to and from file streams. Later in this section, we will use buffered readers (p. 1251) and buffered writers (p. 1250) for reading and writing characters from files, respectively.

Example 20.3 *Copying a File Using a Character Buffer*

Click here to view code image

```
/* Copy a file using a character buffer.
   Command syntax: java CopyCharacterFile <from_file> <to_file> */
import java.io.*;

class CopyCharacterFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileReader fromFile = new FileReader(args[0]);                  // (1)
        FileWriter toFile = new FileWriter(args[1]))   {                // (2)

      // Copy characters using buffer:                                  // (3a)
      char[] buffer = new char[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer characters:
//    fromFile.transferTo(toFile);                                      // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyCharacterFile <from_file> <to_file>"
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

Print Writers

The capabilities of the `OutputStreamWriter` and the `InputStreamReader` classes are limited, as they primarily write and read characters.

In order to write a *text representation* of Java primitive values and objects, a `Print-Writer` should be chained to either a writer, or a byte output stream, or accept a `String` file name, using one of the following constructors:

> Click here to view code image
>
> ```
> PrintWriter(Writer out)
> PrintWriter(Writer out, boolean autoFlush)
> PrintWriter(OutputStream out)
> PrintWriter(OutputStream out, boolean autoFlush)
> PrintWriter(String fileName) throws FileNotFoundException
> PrintWriter(String fileName, Charset charset)
>             throws FileNotFoundException
> PrintWriter(String fileName, String charsetName)
>             throws FileNotFoundException, UnsupportedEncodingException
> ```
>
> The boolean `autoFlush` argument specifies whether the `PrintWriter` should do automatic line flushing.

When the underlying writer is specified, the character encoding supplied by the underlying writer is used. However, an `OutputStream` has no notion of any character encoding, so the necessary intermediate `OutputStreamWriter` is automatically created, which will convert characters into bytes, using the default character encoding.

```
boolean checkError()
protected void clearError()
```

The first method flushes the output stream if it’s not closed and checks its error state.

The second method clears the error state of this output stream.

Writing Text Representation of Primitive Values and Objects

In addition to overriding the `write()` methods from its super class `Writer`, the `Print-Writer` class provides methods for writing text representation of Java primitive values and of objects (see Table 20.6). The `println()` methods write the text representation of their argument to the underlying stream, and then append a *line separator*. The `println()` methods use the correct platform-dependent line separator. For example, on Unix-based platforms the line separator is `'\n'` (newline), while on Windows-based platforms it is `"\r\n"` (carriage return + newline) and on Mac-based platforms it is `'\r'` (carriage return).

Table 20.6 *Print Methods of the* `PrintWriter` *Class*

The `print()` methods	The `println()` methods
	`println()`
`print(boolean b)`	`println(boolean b)`
`print(char c)`	`println(char c)`
`print(int i)`	`println(int i)`
`print(long l)`	`println(long l)`
`print(float f)`	`println(float f)`
`print(double d)`	`println(double d)`
`print(char[] s)`	`println(char[] ca)`
`print(String s)`	`println(String str)`
`print(Object obj)`	`println(Object obj)`

The print methods create a text representation of an object by calling the `toString()` method on the object. The print methods do not throw any `IOException`. Instead, the `checkError()` method of the `PrintWriter` class must be called to check for errors.

Writing Formatted Values

Although formatting of values is covered extensively in Chapter 18, p. 1095, here we mention the support for formatting values provided by I/O streams. The `PrintWriter` class provides the `format()` methods and the `printf()` convenient methods to write *formatted* values. The `printf()` methods are functionally equivalent to the `format()` methods. As the methods return a `PrintWriter`, calls to these methods can be chained.

The `printf()` and the `format()` methods for printing formatted values are also provided by the `PrintStream` and the `Console` classes (p. 1256). The `format()` method is also provided by the `String` class (§8.4, p. 457). We assume familiarity with printing formatted values on the standard output stream by calling the `printf()` method on the `System.out` field which is an object of the `PrintStream` class (§1.9, p. 24).

> Click here to view code image
>
> ```
> PrintWriter format(String format, Object... args)
> PrintWriter format(Locale loc, String format, Object... args)
> PrintWriter printf(String format, Object... args)
> PrintWriter printf(Locale loc, String format, Object... args)
> ```
>
> The `String` parameter `format` specifies how formatting will be done. It contains *format specifiers* that determine how each subsequent value in the variable arity parameter `args` will be formatted and printed. The resulting string from the formatting will be written to the current writer.
>
> If the locale is specified, it is taken into consideration to format the `args`.
>
> Any error in the format string will result in a runtime exception.

Writing Text Files

When writing text representation of values to a file using the default character encoding, any one of the following four procedures for setting up a `PrintWriter` can be used.

Setting up a `PrintWriter` based on an `OutputStreamWriter` which is chained to a `FileOutputStream` (Figure 20.4(a)):

Figure 20.4 *Setting Up a* `PrintWriter` *to Write to a File*

- Create a `FileOutputStream`:

  Click here to view code image

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
```

- Create an `OutputStreamWriter` which is chained to the `FileOutputStream`:

  Click here to view code image

  ```
  OutputStreamWriter outputStream = new OutputStreamWriter(outputFile);
  ```

  The `OutputStreamWriter` uses the default character encoding for writing the characters to the file.

- Create a `PrintWriter` which is chained to the `OutputStreamWriter`:

  Click here to view code image

  ```
  PrintWriter printWriter1 = new PrintWriter(outputStream, true);
  ```

  The value `true` for the second parameter in the constructor will result in the output buffer being flushed by the `println()` and `printf()` methods.

Setting up a `PrintWriter` based on a `FileOutputStream` (Figure 20.4(b)):

- Create a `FileOutputStream`:

  Click here to view code image

  ```
  FileOutputStream outputFile = new FileOutputStream("info.txt");
  ```

- Create a `PrintWriter` which is chained to the `FileOutputStream`:

  Click here to view code image

  ```
  PrintWriter printWriter2 = new PrintWriter(outputFile, true);
  ```

  The intermediate `OutputStreamWriter` to convert the characters using the default encoding is automatically supplied. The output buffer will also perform automatic line flushing.

Setting up a `PrintWriter` based on a `FileWriter` (Figure 20.4(c)):

- Create a `FileWriter` which is a subclass of `OutputStreamWriter`:

  Click here to view code image

```
FileWriter fileWriter = new FileWriter("info.txt");
```

This is equivalent to having an `OutputStreamWriter` chained to a `FileOutputStream` for writing the characters to the file, as shown in Figure 20.4(a).

- Create a `PrintWriter` which is chained to the `FileWriter`:

  Click here to view code image

  ```
  PrintWriter printWriter3 = new PrintWriter(fileWriter, true);
  ```

  The output buffer will be flushed by the `println()` and `printf()` methods.

Setting up a `PrintWriter`, given the file name (Figure 20.4(d)):

- Create a `PrintWriter`, supplying the file name:

  Click here to view code image

  ```
  PrintWriter printWriter4 = new PrintWriter("info.txt");
  ```

  The underlying `OutputStreamWriter` is created to write the characters to the file in the default encoding, as shown in Figure 20.4(d). In this case, there is no automatic flushing by the `println()` and `printf()` methods.

If a specific character encoding is desired for the writer, the third procedure (Figure 20.4(c)) can be used, with the encoding being specified for the `FileWriter`:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileWriter fileWriter = new FileWriter("info.txt", utf8);
PrintWriter printWriter5 = new PrintWriter(fileWriter, true);
```

This writer will use the UTF-8 character encoding to write the characters to the file. Alternatively, we can use a `PrintWriter` constructor that accepts a character encoding:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
PrintWriter printWriter6 = new PrintWriter("info.txt", utf8);
```

A `BufferedWriter` can also be used to improve the efficiency of writing characters to the underlying stream, as explained later in this section.

Reading Text Files

When reading *characters* from a file using the default character encoding, the following two procedures for setting up an `InputStreamReader` can be used.

Setting up an `InputStreamReader` which is chained to a `FileInputStream` (Figure 20.5(a)):

Figure 20.5 *Setting Up Readers to Read Characters*

- Create a `FileInputStream`:

  Click here to view code image

  ```
  FileInputStream inputFile = new FileInputStream("info.txt");
  ```

- Create an `InputStreamReader` which is chained to the `FileInputStream`:

  Click here to view code image

  ```
  InputStreamReader reader = new InputStreamReader(inputFile);
  ```

  The `InputStreamReader` uses the default character encoding for reading the characters from the file.

Setting up a `FileReader` which is a subclass of `InputStreamReader` (Figure 20.5(b)):

- Create a `FileReader`:

  Click here to view code image

  ```
  FileReader fileReader = new FileReader("info.txt");
  ```

  This is equivalent to having an `InputStreamReader` chained to a `FileInputStream` for reading the characters from the file, using the default character encoding.

If a specific character encoding is desired for the reader, the first procedure can be used (Figure 20.5(a)), with the encoding being specified for the `InputStreamReader`:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader = new InputStreamReader(inputFile, utf8);
```

This reader will use the UTF-8 character encoding to read the characters from the file. Alternatively, we can use one of the `FileReader` constructors that accept a character encoding:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileReader reader = new FileReader("info.txt", utf8);
```

A `BufferedReader` can also be used to improve the efficiency of reading characters from the underlying stream, as explained later in this section (p. 1251).

Using Buffered Writers

A `BufferedWriter` can be chained to the underlying writer by using one of the following constructors:

> Click here to view code image
>
> ```
> BufferedWriter(Writer out)
> BufferedWriter(Writer out, int size)
> ```
>
> The default buffer size is used, unless the buffer size is explicitly specified.

Characters, strings, and arrays of characters can be written using the methods for a `Writer`, but these now use buffering to provide efficient writing of characters. In addition, the `BufferedWriter` class provides the method `newLine()` for writing the platform-dependent line separator.

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the UTF-8 character encoding (Figure 20.6(a)):

Figure 20.6 *Buffered Writers*

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileOutputStream   outputFile      = new FileOutputStream("info.txt");
OutputStreamWriter outputStream    = new OutputStreamWriter(outputFile, utf8);
BufferedWriter     bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter        printWriter1    = new PrintWriter(bufferedWriter1, true);
```

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the default character encoding (Figure 20.6(b)):

Click here to view code image

```
FileWriter     fileWriter      = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter    printWriter2    = new PrintWriter(bufferedWriter2, true);
```

Note that in both cases, the `PrintWriter` is used to write the characters. The `Buffered-Writer` is sandwiched between the `PrintWriter` and the underlying `OutputStreamWriter` (which is the superclass of the `FileWriter` class).

Using Buffered Readers

A `BufferedReader` can be chained to the underlying reader by using one of the following constructors:

> Click here to view code image
>
> ```
> BufferedReader(Reader in)
> BufferedReader(Reader in, int size)
> ```
>
> The default buffer size is used, unless the buffer size is explicitly specified.

In addition to the methods of the `Reader` class, the `BufferedReader` class provides the method `readLine()` to read a line of text from the underlying reader.

> Click here to view code image
>
> ```
> String readLine() throws IOException
> ```
>
> The `null` value is returned when the end of the stream is reached. The returned string must explicitly be converted to other values.

The `BufferedReader` class also provides the `lines()` method to create a *stream of text lines* with a buffered reader as the data source (§16.4, p. 902).

> ```
> Stream<String> lines()
> ```
>
> Returns a *finite sequential ordered* `Stream` of element type `String`, where the elements are text lines read by this `BufferedReader`.

The following code creates a `BufferedReader` that can be used to read text lines from a file (Figure 20.7(b)):

Figure 20.7 *Buffered Readers*

Click here to view code image

```
// Using the UTF-8 character encoding:
Charset utf8 = Charset.forName("UTF-8");
FileReader     fileReader      = new FileReader("lines.txt", utf8);
BufferedReader bufferedReader1 = new BufferedReader(reader)

// Use the default encoding:
FileReader     fileReader      = new FileReader("lines.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader);
```

Note that in both cases the `BufferedReader` object is used to read the text lines.

Java primitive values and objects cannot be read directly from their text representation in a file. Characters must be read and converted to the relevant values explicitly. If the text representation of the values is written as *lines of text*, each line can be read and *tokenized* first—that is, grouping characters into *tokens* that meaningfully represent a value. For example, the line `"Potatoes 2.50"` contains two tokens: the item token `"Potatoes"` and the price token `"2.50"`. Once a line is tokenized, the tokens can be parsed to obtain the appropriate type of value. The item token is already of type `String`, but the price token `"2.50"` needs to be parsed to a `double` value using the `Double.parseDouble()` method. A *scanner* provided by the `java.io.Scanner` class or the `String.split()` method called on each line can be used to tokenize the character input—both of which are beyond the scope of this book.

In contrast to Example 20.2, which demonstrated the reading and writing of binary representations of primitive data values, Example 20.4 illustrates the reading and writing of text representations of values using I/O streams that are readers and writers.

The `CharEncodingDemo` class in Example 20.4 writes text representations of values using the UTF-8 character encoding specified at (1). It uses a `try`-with-resources statement for handling the closing of I/O streams, as shown at (2) and (4). The `PrintWriter` is buffered (Figure 20.6(b)). Its underlying writer uses the specified encoding, as shown at (2). Values are written out with the text representation of one value on each line, as shown at (3). The example uses the same character encoding to read the text file. A `BufferedReader` is created (Figure 20.7(b)). Its underlying reader uses the specified encoding, as shown at (4). The text representation of the values is read as one value per line, and parsed accordingly. Each text line in the file is read by calling the `readLine()` method.

The characters in the line are explicitly converted to an appropriate type of value, as shown at (5).

The values are printed on the standard output stream, as shown at (6). We check for the end of the stream at (7), which is signaled by the `null` value returned by the `readLine()` method of the `BufferedReader` class. Note the exceptions that are specified in the `throws` clause of the `main()` method.

Although buffering might seem like overkill in this simple example, for efficiency reasons, it should be considered when reading and writing characters from external storage.

It is a useful exercise to modify Example 20.4 to use the various setups for chaining streams for reading and writing characters, as outlined in this section.

Example 20.4 *Demonstrating Readers and Writers, and Character Encoding*

Click here to view code image

```
import java.io.*;
import java.nio.charset.Charset;
import java.time.LocalDate;

public class CharEncodingDemo {
  public static void main(String[] args)
         throws FileNotFoundException, IOException, NumberFormatException {

    // UTF-8 character encoding.
    Charset utf8 = Charset.forName("UTF-8");                           // (1)

    try(// Create a BufferedWriter that uses UTF-8 character encoding     (2)
        FileWriter writer = new FileWriter("info.txt", utf8);
        BufferedWriter bufferedWriter1 = new BufferedWriter(writer);
        PrintWriter printWriter = new PrintWriter(bufferedWriter1, true);) {

      System.out.println("Writing using encoding: " + writer.getEncoding());
      // Print some values, one on each line.                             (3)
      printWriter.println(LocalDate.now());
      printWriter.println(Integer.MAX_VALUE);
      printWriter.println(Long.MIN_VALUE);
      printWriter.println(Math.PI);
    }

    try(// Create a BufferedReader that uses UTF-8 character encoding     (4)
        FileReader reader = new FileReader("info.txt", utf8);
        BufferedReader bufferedReader = new BufferedReader(reader);) {

      System.out.println("Reading using encoding: " + reader.getEncoding());
      // Read the character input and parse accordingly.                  (5)
      LocalDate ld = LocalDate.parse(bufferedReader.readLine());
      int iMax = Integer.parseInt(bufferedReader.readLine());
      long lMin = Long.parseLong(bufferedReader.readLine());
      double pi = Double.parseDouble(bufferedReader.readLine());
      // Write the values read on the terminal                                 (6)
      System.out.println("Values read:");
      System.out.println(ld);
      System.out.println(iMax);
      System.out.println(lMin);
      System.out.println(pi);

      // Check for end of stream:                                              (7)
      String line = bufferedReader.readLine();
      if (line != null ) {
        System.out.println("More input: " + line);
      } else {
        System.out.println("End of input stream");
      }
    }
  }
}
```

Output from the program:

```
Writing using encoding: UTF8
Reading using encoding: UTF8
Values read:
2021-06-22
2147483647
-9223372036854775808
3.141592653589793
End of input stream
```

The Standard Input, Output, and Error Streams

The *standard output* stream (usually the display) is represented by the `PrintStream` object `System.out`. The *standard input* stream (usually the keyboard) is represented by the `InputStream` object `System.in`. In other words, it is a byte input stream. The *standard error* stream (also usually the display) is represented by `System.err`, which is another object of the `PrintStream` class. The `PrintStream` class offers `print()` methods that act as corresponding `print()` methods from the `PrintWriter` class. The `print()` methods can be used to write output to `System.out` and `System.err`. In other words, both `System.out` and `System.err` act like `PrintWriter`, but in addition they have `write()` methods for writing bytes.

The `System` class provides the methods `setIn(InputStream)`, `setOut(PrintStream)`, and `setErr(PrintStream)` that can be passed an I/O stream to reassign the standard streams.

In order to read *characters* typed by the user, the `Console` class is recommended (p. 1256).

Comparison of Byte Streams and Character

Streams

It is instructive to see which byte streams correspond to which character streams. Table 20.7 shows the correspondence between byte and character streams. Note that not all classes have a corresponding counterpart.

Table 20.7 *Correspondence between Selected Byte and Character Streams*

Byte streams	**Character streams**
`OutputStream`	`Writer`
`InputStream`	`Reader`
No counterpart	`OutputStreamWriter`
No counterpart	`InputStreamReader`
`FileOutputStream`	`FileWriter`
`FileInputStream`	`FileReader`
`BufferedOutputStream`	`BufferedWriter`
`BufferedInputStream`	`BufferedReader`
`PrintStream`	`PrintWriter`
`DataOutputStream`	*No counterpart*
`DataInputStream`	*No counterpart*
`ObjectOutputStream`	*No counterpart*
`ObjectInputStream`	*No counterpart*

20.4 The `Console` Class

A *console* is a unique *character-based* device associated with a JVM. Whether a JVM has a console depends on the platform, and also on the manner in which the JVM is invoked. When the JVM is started from a command line, and the standard input and output streams have not been redirected, the console will normally correspond to the keyboard and the display (Figure 20.8). In any case, the console will be represented by an instance of the class `java.io.Console`. This `Console` instance is a *singleton*, and can only be obtained by calling the static method `console()` of the `System` class. If there is no console associated with the JVM, the `null` value is returned by this method.

Figure 20.8 *Keyboard and Display as Console*

Click here to view code image

```
// Obtaining the console:
Console console = System.console();
if (console == null) {
  System.err.println("No console available.");
  return;
```

```
   }
   // Continue ...
```

For creating dialogue for console-based applications, the `Console` class provides the following functionality:

- Prompt and read a line of character-based response.

  Click here to view code image

  ```
  String username = console.readLine("Enter the username (%d chars): ",
  ```

  The `readLine()` method first prints the formatted prompt on the console, and then returns the characters typed at the console when the line is terminated by the `ENTER` key.

- Prompt and read passwords without echoing the characters on the console.

  Click here to view code image

  ```
  char[] password;
  do {
      password = console.readPassword("Enter password (min. %d chars): "
  } while (password.length < 6);
  ```

  The `readPassword()` method first prints the formatted prompt, and returns the password characters typed by the user in an array of `char` when the line is terminated by the `ENTER` key. The password characters are not echoed on the display.

  Since a password is sensitive data, one recommended practice is to have it stored in memory for only as long as it is necessary and to zero-fill the `char` array as soon as possible in order to overwrite the password characters.

- Print formatted values to the console.

  Similar to the `PrintWriter` and the `PrintStream` classes, the `Console` class also provides the `format()` and the `printf()` methods for printing formatted values, but its methods do *not* allow a locale to be specified.

Note that the console only returns character-based input. For reading other types of values from the standard input stream, the `java.util.Scanner` class can be considered.

The `Console` class provides methods for *formatted prompting* and *reading* from the console, and obtaining the reader associated with it.

> Click here to view code image
>
> ```
> String readLine()
> String readLine(String format, Object... args)
> ```
>
> The first method reads a single line of text from the console. The second method prints a formatted prompt first, then reads a single line of text from the console. The prompt is constructed by formatting the specified `args` according to the specified `format`.
>
> Click here to view code image
>
> ```
> char[] readPassword()
> char[] readPassword(String format, Object... args)
> ```
>
> The first method reads a password or a password phrase from the console with echoing disabled. The second method does the same, but first prints a formatted prompt.
>
> ```
> Reader reader()
> ```
>
> This retrieves the unique `Reader` object associated with this console.

The `Console` class provides the following methods for *writing* formatted strings to the console, and obtaining the writer associated with it:

> Click here to view code image
>
> ```
> Console format(String format, Object... args)
> Console printf(String format, Object... args)
> ```
>
> Write a formatted string to this console's output stream using the specified format string and arguments, according to the default locale. See the `PrintWriter` class with analogous methods (p. 1245).
>
> ```
> PrintWriter writer()
> ```
>
> Retrieves the unique `PrintWriter` object associated with this console.
>
> ```
> void flush()
> ```
>
> Flushes the console and forces any buffered output to be written immediately.

Example 20.5 illustrates using the `Console` class to change a password. The example illustrates the capability of the `Console` class, and in no way should be construed to provide the ultimate secure implementation to change a password.

The console is obtained at (1). The code at (2) implements the procedure for changing the password. The user is asked to submit the new password, and then asked to confirm it. Note that the password characters are not echoed. The respective `char` arrays returned with this input are compared for equality by the static method `equals()` in the `java.util.Arrays` class, which compares two arrays.

Example 20.5 *Changing Passwords*

Click here to view code image

```
import java.io.Console;
import java.io.IOException;
import java.util.Arrays;

/** Class to change the password of a user */
public class ChangePassword {
  public static void main (String[] args) throws IOException  {

    // Obtain the console:                                              (1)
    Console console = System.console();
    if (console == null) {
      System.err.println("No console available.");
      return;
    }

    // Changing the password:                                           (2)
    boolean noMatch = false;
    do {
      // Read the new password and its confirmation:
      char[] newPasswordSelected
          = console.readPassword("Enter your new password: ");
      char[] newPasswordConfirmed
          = console.readPassword("Confirm your new password: ");

      // Compare the supplied passwords:
      noMatch = newPasswordSelected.length == 0  ||
                newPasswordConfirmed.length == 0 ||
                !Arrays.equals(newPasswordSelected, newPasswordConfirmed);
      if (noMatch) {
        console.format("Passwords don't match. Please try again.%n");
      } else {
        // Necessary code to change the password.
        console.format("Password changed.");
      }
    } while (noMatch);
  }
}
```

Running the program:

```
>java ChangePassword
Enter your new password:
Confirm your new password:
Password changed.
```

Review Questions

20.1 Given the following code, under which circumstance will the method return `false`?

Click here to view code image

```
public static boolean test(InputStream is) throws IOException {
  int value = is.read();
  return value >= 0;
}
```

Select the one correct answer.

- **a.** A character of more than 8 bits was read from the input stream.
- **b.** An I/O error occurred.
- **c.** This method will never return `false`.
- **d.** The end of the stream was reached in the input stream.

20.2 How can we programmatically check whether a call to a `print()` method of the `PrintWriter` class was successful or not?

Select the one correct answer.

- **a.** Check if the return value from the call is `-1`.
- **b.** Check if the return value from the call is `null`.
- **c.** Catch the `IOException` that is thrown when an I/O error occurs.
- **d.** Call the `checkError()` method of the `PrintWriter` class immediately after the `print()` method call returns to see if an `IOException` was thrown.

20.3 Given the following program:

Click here to view code image

```
import java.io.*;
public class MoreEndings {
  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("seq.txt");
        InputStreamReader isr = new InputStreamReader(fis);) {
      int i = isr.read();
      while (i != -1) {
        System.out.print((char)i + "|");
        i = isr.read();
      }
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found");
    } catch (EOFException eofe) {
        System.out.println("End of stream");
      } catch (IOException ioe) {
        System.out.println("Input error");
      }
    }
  }
```

Assume that the file `seq.txt` exists in the current directory, has the required access permissions, and contains the string `"Hello"`.

Which statement about the program is true?

Select the one correct answer.

- **a.** The program will fail to compile because a certain unchecked exception is not caught.
- **b.** The program will compile and print `H|e|l|l|o|Input error`.
- **c.** The program will compile and print `H|e|l|l|o|End of stream`.
- **d.** The program will compile, print `H|e|l|l|o|`, and then terminate normally.
- **e.** The program will compile, print `H|e|l|l|o|`, and then block in order to read from the file.
- **f.** The program will compile, print `H|e|l|l|o|`, and terminate because of an uncaught exception.

20.4 Given the following program:

Click here to view code image

```
import java.io.*;

public class NoEndings {
  public static void main(String[] args) {
    try (FileReader fr = new FileReader("greetings.txt");
         BufferedReader br = new BufferedReader(fr)) {
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
    } catch (EOFException eofe) {
      System.out.println("End of stream");
    } catch (IOException ioe) {
      System.out.println("Input error");
    }
  }
}
```

Assume that the file `greeting.txt` exists in the current directory, has the required access permissions, and contains the following two text lines:

```
Hello
Howdy
```

Which statement is true about the program?

Select the one correct answer.

a. The program will fail to compile because the `FileNotFoundException` is not caught.

b. The program will compile, print `Hello|Howdy|null|`, and then terminate normally.

c. The program will compile and print `Hello|Howdy|Input error`.

d. The program will compile and print `Hello|Howdy|End of stream`.

e. The program will compile, print `Hello|Howdy|`, and then block in order to read from the file.

f. The program will compile, print `Hello|Howdy|`, and terminate because of an uncaught exception.

20.5 Object Serialization

Object serialization allows the state of an object to be transformed into a sequence of bytes that can be converted back into a copy of the object (called *deserialization*). After deserialization, the object has the same state as it had when it was serialized, barring any data members that were not serializable. This mechanism is generally known as *persistence*—the serialized result of an object can be stored in a repository from which it can be later retrieved.

Java provides the object serialization facility through the `ObjectInput` and `Object-Output` interfaces, which allow the writing and reading of objects to and from I/O streams. These two interfaces extend the `DataInput` and `DataOutput` interfaces, respectively (Figure 20.1, p. 1235).

The `ObjectOutputStream` class and the `ObjectInputStream` class implement the `Object-Output` interface and the `ObjectInput` interface, respectively, providing methods to write and read binary representation of both objects as well as Java primitive values. Figure 20.9 gives an overview of how these classes can be chained to underlying streams and some selected methods they provide. The figure does not show the methods inherited from the abstract `OutputStream` and `InputStream` superclasses.

Figure 20.9 *Object Stream Chaining*

The read and write methods in the two classes can throw an `IOException`, and the read methods will throw an `EOFException` if an attempt is made to read past the end of the stream.

The `ObjectOutputStream` Class

The class `ObjectOutputStream` can write objects to any stream that is a subclass of the `OutputStream`—for example, to a file or a network connection (socket). An `Object-OutputStream` must be chained to an `OutputStream` using the following constructor:

> Click here to view code image
>
> ```
> ObjectOutputStream(OutputStream out) throws IOException
> ```

For example, in order to store objects in a file and thus provide persistent storage for objects, an `ObjectOutputStream` can be chained to a `FileOutputStream`:

Click here to view code image

```
FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
```

Objects can be written to the stream using the `writeObject()` method of the `Object-OutputStream` class:

> Click here to view code image
>
> ```
> final void writeObject(Object obj) throws IOException
> ```

The `writeObject()` method can be used to write *any* object to a stream, including strings and arrays, as long as the object implements the `java.io.Serializable` interface, which is a *marker interface* with no methods. The `String` class, the primitive wrapper classes, and all array types implement the `Serializable` interface. A serializable object can be any compound object containing references to other objects, and all constituent objects that are serializable are serialized recursively when the compound object is written out. This is true even if there are cyclic references between the objects. Each object is written out only once during serialization. The following information is included when an object is serialized:

- The class information needed to reconstruct the object
- The values of all serializable non-transient and non-static members, including those that are inherited

A checked exception of the type `java.io.NotSerializableException` is thrown if a non-serializable object is encountered during the serialization process. Note also that objects of subclasses that extend a serializable class are always serializable.

The `ObjectInputStream` Class

An `ObjectInputStream` is used to restore (*deserialize*) objects that have previously been serialized using an `ObjectOutputStream`. An `ObjectInputStream` must be chained to an `InputStream`, using the following constructor:

> Click here to view code image
>
> ```
> ObjectInputStream(InputStream in) throws IOException
> ```

For example, in order to restore objects from a file, an `ObjectInputStream` can be chained to a `FileInputStream`:

Click here to view code image

```
FileInputStream inputFile = new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream = new ObjectInputStream(inputFile);
```

The `readObject()` method of the `ObjectInputStream` class is used to read the serialized state of an object from the stream:

> Click here to view code image
>
> ```
> final Object readObject() throws ClassNotFoundException, IOException
> ```

Note that the reference type of the returned object is `Object`, regardless of the actual type of the retrieved object, and can be cast to the desired type. Objects and values must be read in the same order as when they were serialized.

Serializable, non-transient data members of an object, including those data members that are inherited, are restored to the values they had at the time of serialization. For compound objects containing references to other objects, the constituent objects are read to re-create the whole object structure. In order to deserialize objects, the appropriate classes must be available at runtime. Note that new objects are created during deserialization, so that no existing objects are overwritten.

The class `ObjectSerializationDemo` in Example 20.6 serializes some objects in the `writeData()` method at (1), and then deserializes them in the `readData()` method at (2). The `readData()` method also writes the data to the standard output stream.

The `writeData()` method at (1) writes the following values to the output stream: an array of strings (`strArray`), a `long` value (`num`), an array of `int` values (`intArray`), a `String` object (`commonStr`) which is shared with the array `strArray` of strings, and an instance (`oneCD`) of the record class `CD` whose component fields are all serializable.

Duplication is automatically avoided when the same object is serialized several times. The shared `String` object (`commonStr`) is actually only serialized once. Note that the array elements and the characters in a `String` object are not written out explicitly one by one. It is enough to pass the object reference in the `writeObject()` method call. The method also recursively goes through the array of strings, `strArray`, serializing each `String` object in the array. The current state of the `oneCD` instance is also serialized.

The method `readData()` at (2) deserializes the data in the order in which it was written. An explicit cast is needed to convert the reference of a deserialized object to a subtype. Applying the right cast is of course the responsibility of the application. Note that new objects are created by the `readObject()` method, and that an object created during the deserialization process has the same state as the object that was serialized.

Efficient Record Serialization

Example 20.6 shows an example of serializing and deserializing an instance of a record class (§5.14, p. 299). It is worth noting that both processes on records are very efficient as the state components of a record entirely describe the state values to serialize, and as the canonical constructor of a record class is always used to create the complete state of a record, the canonical constructor is also always used during deserialization. By design, selective and customized serialization, discussed later in this section, are not allowed for records.

Example 20.6 *Object Serialization*

Click here to view code image

```
import java.io.Serializable;
import java.time.Year;
/** A record class that represents a CD. */
```

```
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Serializable {
  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

Click here to view code image

```
//Reading and Writing Objects
import java.io.*;
import java.time.Year;
import java.util.Arrays;

public class ObjectSerializationDemo {
  void writeData() {                                    // (1)
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile))

      // Write data:
      String[] strArray = {"Seven", "Eight", "Six"};
      long num = 2014;
      int[] intArray = {1, 3, 1949};
      String commonStr = strArray[2];                    // "Six"
      CD oneCD = new CD("Jaav", "Java Jive", 8, Year.of(2017), CD.Genre.POP);
      outputStream.writeObject(strArray);
      outputStream.writeLong(num);
      outputStream.writeObject(intArray);
      outputStream.writeObject(commonStr);
      outputStream.writeObject(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (IOException e) {
      System.err.println("Write error: " + e);
    }
  }

  void readData() {                                     // (2)
    try (// Set up the input stream:
        FileInputStream inputFile = new FileInputStream("obj-storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read the data:
      String[] strArray = (String[]) inputStream.readObject();
      long num = inputStream.readLong();
      int[] intArray = (int[]) inputStream.readObject();
      String commonStr = (String) inputStream.readObject();
      CD oneCD = (CD) inputStream.readObject();

      // Write data to the standard output stream:
      System.out.println(Arrays.toString(strArray));
      System.out.println(num);
      System.out.println(Arrays.toString(intArray));
      System.out.println(commonStr);
      System.out.println(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (EOFException e) {
      System.err.println("End of stream: " + e);
    } catch (IOException e) {
      System.err.println("Read error: " + e);
      } catch (ClassNotFoundException e) {
        System.err.println("Class not found: " + e);
      }
    }

    public static void main(String[] args) {
      ObjectSerializationDemo demo = new ObjectSerializationDemo();
      demo.writeData();
      demo.readData();
    }
  }
```

Output from the program:

Click here to view code image

```
[Seven, Eight, Six]
2014
[1, 3, 1949]
Six
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

Selective Serialization

As noted earlier, static fields are not serialized, as these are not part of the state of an object. An instance field of an object can be omitted from being serialized by specifying the `transient` modifier in the declaration of the field—typically used for sensitive data in a field. Selective serialization discussed here is *not* applicable to record classes.

Example 20.7 illustrates some salient aspects of serialization. The setup comprises the classes `Wheel` and `Unicycle`, and their client class `SerialClient`. The class `Unicycle` has a field of type `Wheel`, and the class `Wheel` has a field of type `int`. The class `Unicycle` is a compound object with a `Wheel` object as a constituent object. The class `Serial-Client` serializes and deserializes a unicycle in the `try`-with-resources statements at (4) and (5), respectively. The state of the objects is printed to the standard output stream before serialization, and so is the state of the object created by deserialization.

Both the Compound Object and Its Constituents Are Serializable

If we run the program with the following declarations for the `Wheel` and the `Unicycle` classes, where a compound object of the serializable class `Unicycle` uses an object of the serializable class `Wheel` as a constituent object:

Click here to view code image

```
class Wheel implements Serializable {                             // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                          // (2)
  private Wheel wheel;                                            // (3a)
  ...
}
```

we get the following output, showing that both serialization and deserialization were successful:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

A compound object with its constituent objects is often referred to as an *object graph*. Serializing a compound object serializes its complete object graph—that is, the compound object and its constituent objects are recursively serialized.

Example 20.7 *Non-Serializable Objects*

Click here to view code image

```
import java.io.Serializable;

// public class Wheel implements Serializable {                   // (1a)
public class Wheel {                                              // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

Click here to view code image

```
import java.io.Serializable;

public class Unicycle implements Serializable {                    // (2)
  private Wheel wheel;                                             // (3a)
//transient private Wheel wheel;                                   // (3b)

  public Unicycle (Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }
}
```

Click here to view code image

```
import java.io.*;
```

```
public class SerialClient {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {

    try (// Set up the output stream:                                     // (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile))

      // Write the data:
      Wheel wheel = new Wheel(65);
      Unicycle uc = new Unicycle(wheel);
      System.out.println("Before writing: " + uc);
      outputStream.writeObject(uc);
    }

    try (// Set up the input streams:                                     // (5)
        FileInputStream inputFile = new FileInputStream("storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read data.
      Unicycle uc = (Unicycle) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + uc);
    }
  }
}
```

Transient Fields Are Not Serializable

If we declare the `wheel` field of the `Unicycle` class in Example 20.7 to be `transient` for the sake of demonstrating how such fields are handled at serialization, (3b):

Click here to view code image

```
class Wheel implements Serializable {                                     // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                                  // (2)
  transient private Wheel wheel;                                          // (3b)
  ...
}
```

we get the following output, showing that the `wheel` field of the `Unicycle` object was not serialized:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with null
```

Serializable Compound Object with Non-Serializable Constituents

In order for a compound object to be serialized, its constituent objects must be serializable; otherwise, serialization will not succeed. If the class `Wheel` in Example 20.7 is *not* serializable, (1b):

Click here to view code image

```
class Wheel {                                                    // (1b)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                         // (2)
  private Wheel wheel;                                           // (3a)
  ...
}
```

we get the following output when we run the program—that is, a `Unicycle` object *cannot* be serialized because its constituent `Wheel` object is not serializable:

Click here to view code image

```
>java SerialClient
Before writing: Unicycle with wheel size: 65
Exception in thread "main" java.io.NotSerializableException: Wheel
  ...
  at SerialClient.main(SerialClient.java:20)
```

Customizing Object Serialization

As we have seen, the class of the object must implement the `Serializable` interface if we want the object to be serialized. If this object is a compound object, then all its constituent objects must also be serializable, and so on.

It is not always possible for a client to declare that a class is `Serializable`. It might be declared `final`, and therefore not extendable. The client might not have access to the code, or extending this class with a serializable subclass might not be an option. Java provides a customizable solution for serializing objects in such cases.

Customized serialization discussed here is *not* applicable to record classes.

The basic idea behind the scheme is to use default serialization as much as possible, and to provide *hooks* in the code for the serialization mechanism to call specific methods to deal with objects or values that should not or cannot be serialized by the default methods of the object streams.

Customizing serialization is illustrated in Example 20.8, using the

`Wheel` and `Unicycle` classes from Example 20.7. The serializable class `Unicycle` would like to use the `Wheel` class, but this class is not serializable. If the `wheel` field in the `Unicycle` class is declared to be `transient`, it will be ignored by the default serialization procedure. This is not a viable option, as the unicycle will be missing the wheel size when a serialized unicycle is deserialized, as was illustrated in Example 20.7.

Any serializable object has the option of customizing its own serialization if it implements the following pair of methods:

Click here to view code image

```
private void writeObject(ObjectOutputStream) throws IOException;
private void readObject(ObjectInputStream)
                    throws IOException, ClassNotFoundException;
```

These methods are *not* part of any interface. Although private, these methods can be called by the JVM. The first method above is called on the object when its serialization starts. The serialization procedure uses the reference value of the object to be serialized that is passed in the call to the `ObjectOutputStream.writeObject()` method, which in turn calls the first method above on this object. The second method above is called on the object created when the deserialization procedure is initiated by the call to the `ObjectInputStream.readObject()` method.

Customizing serialization for objects of the class `Unicycle` in Example 20.8 is achieved by the private methods at (3c) and (3d). Note that the field `wheel` is declared `transient` at (3b) and excluded by the normal serialization process.

In the private method `writeObject()` at (3c) in Example 20.8, the pertinent lines of code are the following:

Click here to view code image

```
oos.defaultWriteObject();               // Method in the ObjectOutputStream cl
oos.writeInt(wheel.getWheelSize());     // Method in the ObjectOutputStream cl
```

The call to the `defaultWriteObject()` method of the `ObjectOutputStream` does what its name implies: normal serialization of the current object. The second line of code does the customization: It writes the binary `int` value of the wheel size to the `ObjectOutputStream`. The code for customization can be called both before and after the call to the `defaultWriteObject()` method, as long as the same order is used during deserialization.

In the private method `readObject()` at (3d), the pertinent lines of code are the following:

Click here to view code image

```
ois.defaultReadObject();              // Method in the ObjectInputStream cla
int wheelSize = ois.readInt();        // Method in the ObjectInputStream cla
this.wheel = new Wheel(wheelSize);
```

The call to the `defaultReadObject()` method of the `ObjectInputStream` does what its name implies: normal deserialization of the current object. The second line of code reads the binary `int` value of the wheel size from the `ObjectInputStream`. The third line of code creates a `Wheel` object, passes this value in the constructor call, and assigns its reference value to the `wheel` field of the current object. Again, code for customization can be called both before and after the call to the `defaultReadObject()` method, as long as it is in correspondence with the customization code in the `writeObject()` method.

The client class `SerialClient` in Example 20.8 is the same as the one in Example 20.7. The output from the program confirms that the object state prior to serialization is identical to the object state after deserialization.

Example 20.8 *Customized Serialization*

Click here to view code image

```
public class Wheel {                                          // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  public int getWheelSize() { return wheelSize; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

Click here to view code image

```
import java.io.*;

public class Unicycle implements Serializable {               // (2)
  transient private Wheel wheel;                              // (3b)

  public Unicycle(Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }

  private void writeObject(ObjectOutputStream oos) {          // (3c)
    try {
      oos.defaultWriteObject();
      oos.writeInt(wheel.getWheelSize());
    } catch (IOException e) {
      e.printStackTrace();
      }
    }

    private void readObject(ObjectInputStream ois) {                  // (3d)
      try {
        ois.defaultReadObject();
        int wheelSize = ois.readInt();
        this.wheel = new Wheel(wheelSize);
      } catch (IOException e) {
        e.printStackTrace();
      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      }
    }
  }
```

Click here to view code image

```
public class SerialClient { // Same as in Example 20.7 }
```

Output from the program:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

Serialization and Inheritance

The inheritance hierarchy of an object also determines what its state will be after it is deserialized. An object will have the same state at deserialization as it had at the time it was serialized if *all* its superclasses are also serializable. This is because the normal object creation and initialization procedure using constructors is *not* run during deserialization.

However, if any superclass of an object is *not* serializable, then the normal creation procedure using *no-argument* or *default* constructors *is* run, starting at the first non-serializable superclass, all the way up to the `Object` class. This means that the state at deserialization might not be the same as at the time the object was serialized because superconstructors run during deserialization may have initialized the state of the object. If the non-serializable superclass does not provide a non-argument constructor or the default constructor, a `java.io.InvalidClassException` is thrown during deserialization.

Example 20.9 illustrates how inheritance affects serialization. The `Student` class is a subclass of the `Person` class. Whether the superclass `Person` is serializable or not has implications for serializing objects of the `Student` subclass, in particular, when their byte representation is deserialized.

Superclass Is Serializable

If the superclass is serializable, then any object of a subclass is also serializable. In Example 20.9, the code at (4) in the class `SerialInheritance` serializes a `Student` object:

Click here to view code image

```
Student student = new Student("Pendu", 1007);
System.out.println("Before writing: " + student);
outputStream.writeObject(student);
```

The corresponding code for deserialization of the streamed object is at (5) in the class `SerialInheritance`:

Click here to view code image

```
Student student = (Student) inputStream.readObject();
System.out.println("After reading: " + student);
```

We get the following output from the program in Example 20.9 when it is run with (1a) and (3a) in the `Person` class and the `Student` class, respectively—that is, when the superclass is serializable and so is the subclass, by virtue of inheritance.

The results show that the object state prior to serialization is identical to the object state after deserialization. In this case, no superclass constructors were run during deserialization.

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
After reading: Student state(Pendu, 1007)
```

Superclass Is Not Serializable

However, the result of deserialization is not the same when the superclass `Person` is not serializable, but the subclass `Student` is. We get the following output from the program in Example 20.9 when it is run with (1b) and (3b) in the `Person` class and the `Student` class, respectively—that is, when only the subclass is serializable, but not the superclass. The output shows that the object state prior to serialization is not identical to the object state after deserialization.

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
No-argument constructor executed.
After reading: Student state(null, 1007)
```

During deserialization, the *zero-argument* constructor of the `Person` superclass at (2) is run. As we can see from the declaration of the `Person` class in Example 20.9, this zero-argument constructor does not initialize the `name` field, which remains initialized with the default value for reference types (`null`).

If the superclass `Person` does not provide the no-argument constructor or the default constructor, as in the declaration below, the call to the `readObject()` method to perform deserialization throws an `InvalidClassException`.

Click here to view code image

```
public class Person  {                                              // (1b)
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
Exception in thread "main" java.io.InvalidClassException:
        Student; no valid constructor
        ...
        at SerialInheritance.main(SerialInheritance.java:28)
```

The upshot of serializing objects of subclasses is that the superclass should be serializable, unless there are compelling reasons for why it is not. And if the superclass is not serializable, it should at least provide either the default constructor or the no-argument constructor to avoid an exception during deserialization.

Although a superclass might be serializable, its subclasses can prevent their objects from being serialized by implementing the `private` method `writeObject (ObjectOutputStream)` that throws a `java.io.NotSerializableException`.

Example 20.9 *Serialization and Inheritance*

Click here to view code image

```
import java.io.Serializable;

// A superclass
public class Person implements Serializable {                       // (1a)
//public class Person  {                                            // (1b)
  private String name;
  public Person() {                                            // (2)
    System.out.println("No-argument constructor executed.");
  }
  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

Click here to view code image

```
import java.io.Serializable;

public class Student extends Person {                            // (3a)
//public class Student extends Person implements Serializable {  // (3b)

  private long studNum;

  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }

  @Override
  public String toString() {
    return "Student state(" + getName() + ", " + studNum + ")";
  }
}
```

Click here to view code image

```
import java.io.*;

public class SerialInheritance {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {

    // Serialization:
    try (// Set up the output stream:                                         (
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile))

      // Write data:
      Student student = new Student("Pendu", 1007);
      System.out.println("Before writing: " + student);
      outputStream.writeObject(student);
    }

    // Deserialization:
    try (// Set up the input stream:
        FileInputStream inputFile = new FileInputStream("storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read data.
      Student student = (Student) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + student);
    }
  }
}
```

Serialization and Versioning

Class versioning comes into play when we serialize an object with one definition of a class, but deserialize the streamed object with a different class definition. By streamed object, we mean the serialized representation of an object. Between serialization and deserialization of an object, the class definition can change.

Note that at serialization and at deserialization, the definition of the class (i.e., its bytecode file) should be accessible. In the examples so far, the class definition has been the same at both serialization and deserialization. Example 20.10 illustrates the problem of class definition mismatch at deserialization and the solution provided by Java.

Example 20.10 makes use of the following classes (numbering refers to code lines in the example):

(1) The original version of the serializable class `Item`. It has one field named `price`. An object of this class will be serialized and read using different versions of this class.

(2) A newer version of the class `Item` that has been augmented with a field for the weight of an item. This class will only be used for deserialization of objects that have been serialized with the original version of the class.

(3) The class `Serializer` serializes an object of the original version of the class `Item`.

(4) The class `DeSerializer` deserializes a streamed object of the class `Item`. In the example, deserialization is based on a different version of the `Item` class than the one used at serialization.

There are no surprises if we use the original class `Item` to serialize and deserialize an object of the `Item` class.

Click here to view code image

```
// Original version of the Item class.
public class Item implements Serializable {                          // (1)
  private double price;
//...
}
```

Result:

```
Before writing: Price: 100.00
After reading: Price: 100.00
```

If we deserialize a streamed object of the original class `Item` at (1) based on the byte-code file of the augmented version of the `Item` class at (2), an `InvalidClassException` is thrown at runtime.

Click here to view code image

```
// New version of the Item class.
public class Item implements Serializable {                            // (2)
  private double price;
  private double weight;                                    // Additional field
//...
}
```

Result (*edited to fit on the page*):

Click here to view code image

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = -4194294879924868414,
  local class serialVersionUID = -1186964199368835959
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

The question is, how was the class definition mismatch discovered at runtime? The answer lies in the stack trace of the exception thrown. The local class was incompatible, meaning the class we are using to deserialize is not compatible with the class that was used when the object was serialized. In addition, two long numbers are printed, representing the `serialVersionUID` of the respective class definitions. The first `serialVersionUID` is generated by the serialization process based on the class definition and becomes part of the streamed object. The second `serialVersionUID` is generated based on the local class definition that is accessible at deserialization. The two are not equal, and deserialization fails.

A serializable class can provide its `serialVersionUID` by declaring it in its class declaration, exactly as shown below, except for the initial value which of course can be different:

Click here to view code image

```
static final long serialVersionUID = 100L;            // Appropriate value.
```

As we saw in the example above, if a serializable class does not provide a `serialVersionUID`, one is implicitly generated. By providing an explicit `serialVersionUID`, it is possible to control what happens at deserialization. As newer versions of the class are created, the `serialVersionUID` can be kept the same until it is deemed that older streamed objects are no longer compatible for deserialization. After the change to the `serialVersionUID`, it will not be possible to deserialize older streamed objects of the class based on newer versions of the class. Although static members of a class are not serialized, the only exception is the value of the `static final long serialVersionUID` field.

In the scenario below, the original version and the newer version of the class `Item` both declare a `serialVersionUID` at (1a) and at (2a), respectively, that has the same value. An `Item` object is serialized using the original version, but deserialized based on the newer version. We see that serialization succeeds, and the `weight` field is initialized to the default value `0.0`. In other words, the object created is of the newer version of the class.

Click here to view code image

```
// Original version of the Item class.
public class Item implements Serializable {    // (1)
  static final long serialVersionUID = 1000L;  // (1a)

  private double price;
//...
}

// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1000L;  // (2a) Same serialVersionUID
  private double price;
  private double weight;                       // Additional field
//...
}
```

Result:

Click here to view code image

```
Before writing: Price: 100.00
After reading: Price: 100.00, Weight: 0.00
```

However, if we now deserialize the streamed object of the original class having `1000L` as the `serialVersionUID`, based on the newer version of the class having the `serialVersionUID` equal to `1001L`, deserialization fails as we would expect because the `serialVersionUID`s are different.

Click here to view code image

```
// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1001L;  // (2b) Different serialVersion
  private double price;
  private double weight;
//...
}
```

Result (*edited to fit on the page*):

Click here to view code image

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = 1000,
  local class serialVersionUID = 1001
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

Best practices advocate that serializable classes should use the `serialVersionUID` solution for better control of what happens at deserialization as classes evolve.

Example 20.10 *Class Versioning*

Click here to view code image

```
import java.io.Serializable;

// Original version of the Item class.
public class Item implements Serializable {                                    // (1)
//static final long serialVersionUID = 1000L;                                  // (1a)

  private double price;

  public Item(double price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return String.format("Price: %.2f%n", this.price);
  }
}
```

Click here to view code image

```
import java.io.Serializable;

// New version of the Item class.
public class Item implements Serializable {                                    // (2)
//static final long serialVersionUID = 1000L;                                  // (2a)
//static final long serialVersionUID = 1001L;                                  // (2b)

  private double price;
  private double weight;
  public Item(double price, double weight) {
    this.price = price;
    this.weight = weight;
  }
  @Override
  public String toString() {
    return String.format("Price: %.2f, Weight: %.2f", this.price, this.weight)
  }
}
```

Click here to view code image

```
// Serializer for objects of class Item.
import java.io.*;

public class Serializer {                                                        // (3
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("item_storage.dat")
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile))

      // Serialize an object of the original class:
      Item item = new Item(100.00);
      System.out.println("Before writing: " + item);
      outputStream.writeObject(item);
    }
  }
}
```

Click here to view code image

```
// Deserializer for objects of class Item.
import java.io.*;

public class DeSerializer{                                               // (4
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the input streams:
        FileInputStream inputFile = new FileInputStream("item_storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read a serialized object of the Item class.
      Item item = (Item) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + item);
    }
  }
}
```

Review Questions

20.5 Which of the following best describes the data written by an `ObjectOutputStream`?

Select the one correct answer.

a. Bytes and other Java primitive types

b. Object graphs

c. Object graphs and Java primitive types

d. Single objects

e. Single objects and Java primitive types

20.6 Which of the following statements are true about serialization? Select the four correct answers.

a. All static fields of a class are treated the same way as `transient` instance fields of the class when it comes to serialization.

b. All instance fields of a serializable class must also be serializable or be specified with the `transient` modifier.

c. Any `private` instance field of a class is not serialized.

d. A serializable class must implement the `Serializable` interface.

e. A serializable class must extend the `Serializable` class.

f. Subclasses of a serializable class must explicitly extend the `Serializable` class in order to serialize subclass objects.

g. Subclasses of a serializable class must explicitly implement the `Serializable` interface in order to serialize subclass objects.

h. A serializable class must be specified with the `final` modifier.

i. A `serialVersionUID` is always stored in the streamed object of a serializable class.

20.7 Which of the following statements is true?

Select the one correct answer.

a. All versions of a serializable class must provide a declaration of a `serialVersionUID`.

b. Even if two versions of a serializable class have the same `serialVersionUID`, there is no guarantee that an object serialized based on one version can be deserialized to an object of the other.

c. The `serialVersionUID` of any two unrelated serializable classes must be unique.

d. The `serialVersionUID` of a serializable class must be incremented every time a new version of the class is created.

e. Only a serializable class can declare a `static final` field of type `long` having the name `serialVersionUID`.

20.8 Given the following code:

Click here to view code image

```
public class Person  {
  protected String name;
  public Person() {this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
import java.io.Serializable;
public class Student extends Person implements Serializable {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
```

```
import java.io.*;
public class RQ800_10 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat
         ObjectOutputStream outputStream = new ObjectOutputStream(output
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

Which statement about the program is true?

Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100)(Aesop, 100)`.

d. It will print `(Aesop, 100)(null, 100)`.

e. It will print `(Aesop, 100)(NoName, 100)`.

20.9 Given the following code:

Click here to view code image

```
import java.io.Serializable;
public class Person implements Serializable {
  protected String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
public class Student extends Person {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
```

```
import java.io.*;
public class RQ800_30 {
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat
         ObjectOutputStream outputStream = new ObjectOutputStream(output
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

Which statement about the program is true?
Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100)(Aesop, 100)`.

d. It will print `(Aesop, 100)(null, 100)`.

e. It will print `(Aesop, 100)(NoName, 100)`.

20.10 Given the following code:

Click here to view code image

```
public class Person {
  protected transient String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
public class Student extends Person {
  protected long studNum;
  public Student() { }
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class GraduateStudent extends Student implements Serializable {
  private int year;
  public GraduateStudent(String name, long studNum, int year) {
    super(name, studNum);
    this.year = year;
  }

  public String toString() {
    return "(" + name + ", " + studNum + ", " + year  + ")";
  }

  private void readObject(ObjectInputStream ois)
      throws IOException, ClassNotFoundException {
    ois.defaultReadObject();
    this.name = "NewName";
    this.studNum = 200;
    this.year = 2;
  }
}
```

```
import java.io.*;
public class RQ800_70 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat
         ObjectOutputStream outputStream = new ObjectOutputStream(output
      GraduateStudent stud1 = new GraduateStudent("Aesop", 100, 1);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile
      GraduateStudent stud2 = (GraduateStudent) inputStream.readObject()
      System.out.println(stud2);
    }
  }
}
```

Which statement about the program is true?
Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100, 1)(Aesop, 100, 1).`

d. It will print `(Aesop, 100, 1)(NoName, 0, 1).`

e. It will print `(Aesop, 100, 1)(NewName, 200, 2).`

20.11 Given the following code:

Click here to view code image

```
import java.io.Serializable;
public class Product implements Serializable {
  private String name;

  public Product(String name) {
    this.name = name;
    System.out.print("product ");
  }

  @Override
  public String toString() {
    return name;
  }
}
```

```
public class Food extends Product {
  private int calories;

  public Food(String name, int calories) {
    super(name);
    this.calories = calories;
    System.out.print("food ");
  }

  @Override
  public String toString() {
    return super.toString()+" "+calories;
  }
}
```

```
import java.io.*;
public class RQ20 {
  public static void main(String[] args) {

    Product p = new Food("cookie", 300);
    try(ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream("prod.dat"))) {
      out.writeObject(p);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try(ObjectInputStream in =
            new ObjectInputStream(new FileInputStream("prod.dat"))) {
      p = (Food)in.readObject();
    } catch (Exception ex) {
      System.out.println("error deserializing product");
    }
    System.out.println(p);
  }
}
```

What is the result?

Select the one correct answer.

- **a.** `product food cookie 300`
- **b.** `product food product food cookie 300`
- **c.** `product food cookie 0`
- **d.** `product food product food cookie 0`
- **e.** `product food error serializing product`
- **f.** `product food error deserializing product`

20.12 Given the following code:

Click here to view code image

```
import java.io.*;
public class RQ21 {
  public static void main(String[] args) {
    char[] buffer = new char[4];
    int count = 0;
    try(FileReader in = new FileReader("test1.txt");
        FileWriter out = new FileWriter("test2.txt")) {
      while((count = in.read(buffer)) != -1) {
        out.write(buffer);
      }
    } catch (Exception ex) {
      System.out.println("error");
    }
  }
}
```

Assume that the `text1.txt` file only contains the line `"abcdefg"`. What is the content of the `text2.txt` file after the program is run?

Select the one correct answer.

- **a.** `abcdefg`
- **b.** `abcd`
- **c.** `abcdefgd`
- **d.** `abcdef`

20.13 Given the following code:

Click here to view code image

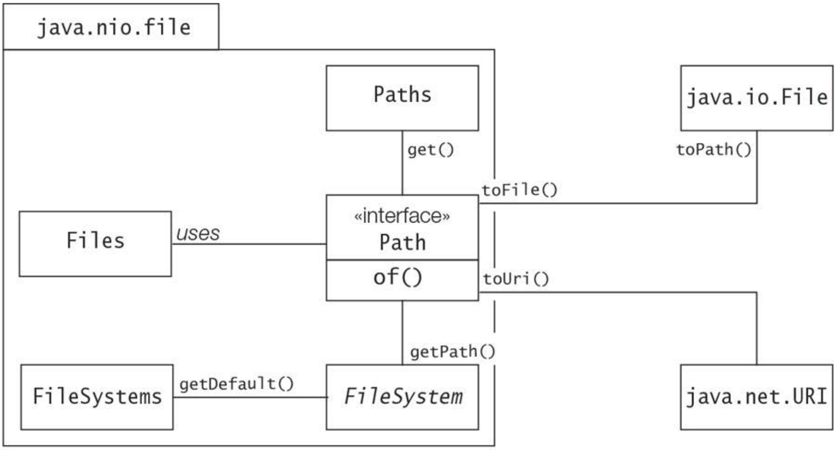
```
import java.io.*;
public class Album implements Serializable {
  private static int numberOfTracks = 5;

  private String title;
  private transient int currentTrack;

  public Album(String title, int currentTrack) {
    this.title = title;
    this.currentTrack = currentTrack;
  }

  public void readObject(ObjectInputStream in)
                              throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    currentTrack = 3;
  }
  @Override
  public String toString() {
    return title+" "+numberOfTracks +" "+currentTrack;
  }
}
```

```
import java.io.*;
public class RQ22 {
  public static void main(String[] args) {
    Album a = new Album("Songs", 2);
    try (ObjectOutputStream out =
              new ObjectOutputStream(new FileOutputStream("song.dat"))) {
      out.writeObject(a);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try (ObjectInputStream in =
              new ObjectInputStream(new FileInputStream("song.dat"))) {
      a = (Album)in.readObject();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    System.out.println(a);
  }
}
```

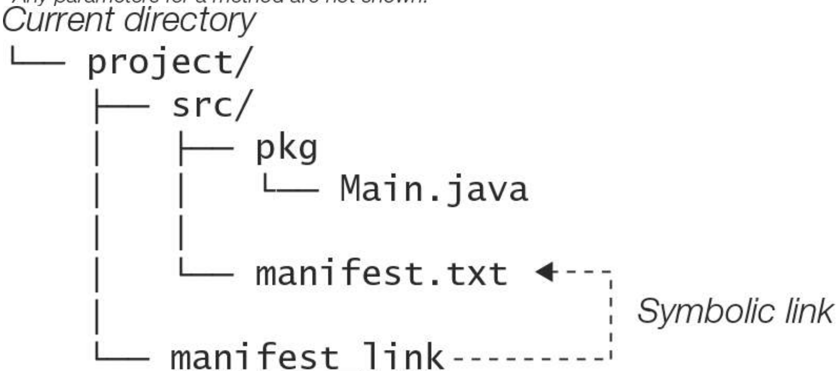
What is the result?

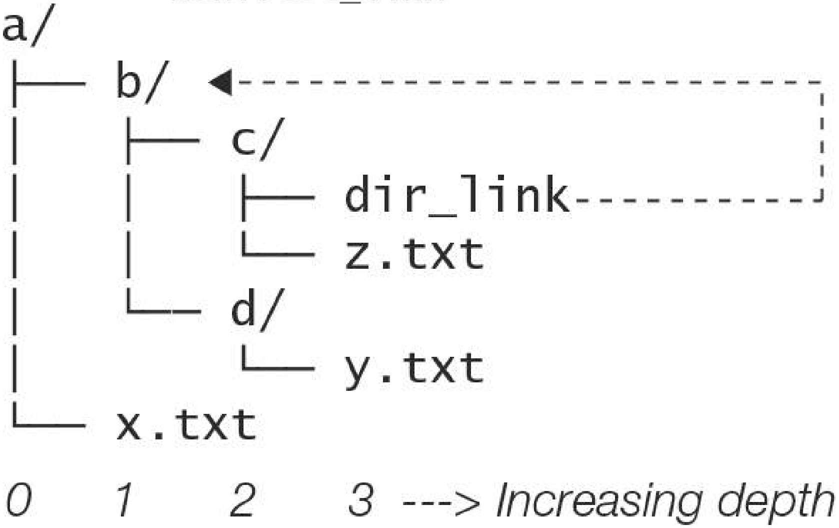
Select the one correct answer.

a. `Songs 5 3`
b. `Songs 5 2`
c. `Songs 5 0`
d. `Songs 0 3`
e. `Songs 0 2`
f. `Songs 0 0`


java.nio.file
Paths
get()
java.io.File
toPath()
toFile()
Files
uses
«interface»
Path
of()
toUri()
getPath()
FileSystems
getDefault()
FileSystem
java.net.URI

Any parameters for a method are not shown.


Current directory
project/
src/
pkg
Main.java
manifest.txt
manifest_link
Symbolic link


a/
b/
c/
dir_link
z.txt
d/
y.txt
x.txt
0
1
2
3
---> Increasing depth

Java I/O: Part II

21

Chapter Topics

- Understanding the characteristics of a hierarchical file system
- Creating `Path` objects with the `of()` method of the `Path` interface, the `getPath()` method of the default file system, and the `get()` method of the `Paths` class
- Interoperability of `Path` objects with the `java.io.File` legacy class and the `java.net.URI` class
- Querying `Path` objects: converting, normalizing, resolving, relativizing, and comparing
- Using methods of the `Path` interface to perform operations on directory entries: Existence, uniqueness, copying, moving, and renaming directory entries
- Reading and writing byte and character data to files using `Path` objects and buffered I/O streams
- Understanding the purpose of specifying a variable arity parameter to customize the behavior of file operations with constants defined by the `LinkOption`, `StandardCopyOption`, `OpenOption`, and `FileVisitOption` enum types in the `java.nio.file` package
- Using methods of the `Files` class for accessing specific file attributes: size, kind of entry, file accessibility, last modified timestamp, owner, file permissions, and access by attribute name
- Retrieving read-only file attributes in a bulk operation provided by the `Files.readAttributes()` method, and

accessing them using the interfaces `BasicFileAttributes`, `PosixFileAttributes`, and `DosFileAttributes`

- Retrieving readable and updatable file attributes in a bulk operation provided by the `Files.getFileAttributeView()` method that creates *views* represented by the interfaces `BasicFileAttributeView`, `PosixFileAttributeView`, and `DosFileAttributeView`
- Creating directory entries with methods provided by the `Files` class: regular and temporary files, regular and temporary directories, and symbolic links
- Using a stream to read text files (`Files.lines()`)
- Using streams for listing entries in a directory (`Files.list()`), for walking the directory hierarchy (`Files.walk()`), and for finding directory entries (`Files.find()`)

Java SE 17 Developer Exam O	
[9.3] Create, traverse, read, and write Path objects and their properties using java.nio.file API	*§21.1, p. 1287* *to* *§21.8, p. 1345*

Java SE 11 Developer Exam O	
[9.3] Handle file system objects using java.nio.file API	*§21.1, p. 1287* *to* *§21.8, p. 1345*

The standard I/O API in the `java.io` package provides the capabilities for reading and writing various kinds of data using the concept of I/O streams (Chapter 20, p. 1231). However, its support for *managing* files and directories in a file system has been superceded by the newer and enhanced I/O API called *NIO.2* (*Non-blocking I/O,* version 2) in the `java.nio` package.

The primary focus in this chapter is on the support provided by the NIO.2 API for programmatically interacting with a file system for accessing files, managing file attributes, and traversing the file system. Primary support for file I/O and accessing files and directories in a file system is provided by the `java.nio.file` package.

The `java.nio.file.Path` interface allows paths in the file system to be represented programmatically. It provides extensive support for constructing and querying paths, and for how paths can be used to manipulate files and directories in the file system.

The `java.nio.file.Files` utility class provides comprehensive support for file operations, such as querying, creating, deleting, copying, and moving files. In addition, it provides methods for creating streams on files and directories, making it possible to write powerful file operations.

Support for interoperability between the two I/O APIs is also covered, and other auxiliary classes are introduced where they are needed in the chapter.

21.1 Characteristics of a Hierarchical File System

We first look at some characteristics of a hierarchical file system, and introduce the terminology used in this chapter.

Hierarchical File Systems

A *file system* allows persistent storage and organization of data as a *hierarchical* (or *tree*) structure on some external media. A tree structure has a *root component* (also called *root node*) at the top, under which other tree nodes represent files and directories. Each directory can have files and nested directories (often called *subdirectories*). An operative system can have multiple file systems, each of which is identifiable by its root component.

Below is an example of a file system, where the root component is platform dependent. On Unix-based platforms, the root component of the file system is denoted by the slash character (`/`). The root component for Windows-based platforms is a combination of a *volume* name (i.e., file system name), a colon (`:`), and a backslash character (`\`). For example, `C:\` designates the root component of the ubiquitous volume named `C` on a Windows-based platform.

Click here to view code image

```
root_component

a
    b
        c
            d

... ... ... ...
```

When it is not necessary to distinguish between a file and a directory in the file system, we will use the term *directory entry* to mean both. Each directory entry in a hierarchical file system is uniquely identifiable by a *path* from the root component to the node representing the directory entry. The path of a directory entry in a file system is specified using the naming conventions of the host system, where *name elements* that comprise the path are separated by a platform-specific *name separator character* or *delimiter*. The name separator character for name elements on Unix-based platforms is the slash character (`/`), whereas on Windows-based platforms, it is the backslash character (`\`). For the most part, we will use conventions for Unix-based platforms.

Two examples of paths are given below, where each path has three name elements.

Click here to view code image

```
/a/b/c           on Unix-based platforms
C:\a\b\c         on Windows-based platforms
```

Absolute and Relative Paths

Directory entries can be referenced using both *absolute* and *relative* paths, but the path naming must follow the conventions of the host platform.

An absolute path starts with the platform-dependent root component of the file system, as in the examples above. All information is contained in an absolute path to reference the directory entry—that is, an absolute path uniquely identifies a directory entry in the file system.

A relative path is without designation of the root component and therefore requires additional path information to locate its directory entry in the file system. Typically, a relative path is interpreted in relation to the *current directory* (see the next subsection). Some examples of relative paths are given below, but they alone are not enough to uniquely identify a directory entry.

Click here to view code image

```
c/d                 on Unix-based platforms
c\d                 on Windows-based platforms
```

Note that in a path, the name elements are all parent directory names, except for the last name element, which can be either a file name or a directory name. The name of a directory entry does not distinguish whether it is a file or a directory in the file system. Although *file extensions* can be used in a file name for readability, it is immaterial for the file system. Care must also be exercised when choosing name elements, as characters allowed are usually platform dependent.

Java programs should not rely on system-specific path conventions. In the next section we will construct paths in a platform-independent way.

Current and Parent Directory Designators

A file system has a notion of a current directory which changes while traversing in the file system. The *current directory* is designated by the period character (`.`) and its *parent directory* by two periods (`..`). These designators can be used in constructing paths. Given that the current directory is `/a/b` in the directory tree shown earlier, Table 21.1 illustrates relative paths constructed using the current and the parent directory designators, and their corresponding absolute paths.

Table 21.1 *Using Current and Parent Directory Designators*

Relative path (Current directo	Absolute path
`./c/d`	`/a/b/c/d`
`.`	`/a/b`
`./..`	`/a`
`./../..`	`/` (i.e., the root component)

Symbolic Links

Apart from regular files, a file system may allow symbolic links to be created. A *symbolic link* (also called *a hard link*, *a shortcut*, or *an alias*) is a special file that acts as a reference to a directory entry, called the *target* of the symbolic link. Creating an *alias* of a directory entry in the file system is similar to creating a symbolic link. Using symbolic links is transparent to the user, as the file system takes care of using the target when a symbolic link is used in a file operation, with one caveat: Deleting the symbolic link does *not* delete the target. In many file operations, it is possible to indicate whether symbolic links should be followed or not.

21.2 Creating `Path` Objects

File operations that a Java program invokes are performed on a file system. Unless the program utilizes a specific file system constructed by the factory methods of the `java.nio.file.FileSystems` utility class, file operations are performed on a file system called the *default file system* that is accessible to the JVM. We can think of the default file system as the *local* file system. The default file system is platform specific and can be obtained as an instance of the `java.nio.file.FileSystem` abstract class by invoking the `getDefault()` factory method of the `FileSystems` utility class.

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

File operations can access the default file system directly to perform their operations. The default file system can be queried for various file system properties.

> Click here to view code image
>
> `static FileSystem getDefault()` Declared in `java.nio.file.FileSystems`
>
> Returns the platform-specific default file system. If the method is called several times, it returns the same `FileSystem` instance for the default file system.
>
> Click here to view code image
>
> `abstract String getSeparator()` Declared in `java.nio.file.FileSystem`
>
> Returns the platform-specific name separator for name elements in a path, represented as a string.

Paths in the file system are programmatically represented by objects that implement the `Path` interface. The `Path` interface and various other classes provide factory methods that create `Path` objects (see below). A `Path` object is also platform specific.

A `Path` object implements the `Iterable<Path>` interface, meaning it is possible to traverse over its name elements from the first name element to the last. It is also immutable and therefore thread-safe.

In this section, we look at how to create `Path` objects. A `Path` object can be queried and manipulated in various ways, and may not represent an existing directory entry in the file system (p. 1294). A `Path` object can be used in file operations to access and manipulate the directory entry it denotes in the file system (p. 1304).

The methods below can be used to create `Path` objects. The methods are also shown in Figure 21.1, together with the relationship between the different classes of these methods. Note in particular the interoperability between the `java.nio.file.Path` interface that represents a path and the two classes `java.io.File` and `java.net.URI`. The class `java.io.File` represents a pathname in the standard I/O API and the class `java.net.URI` represents a *Uniform Resource Identifier* (*URI*) that identifies a resource (e.g., a file or a website).

> Click here to view code image
>
> ```
> static Path of(String first, String... more)
> static Path of(URI uri)
> ```
>
> These methods are declared in the `java.nio.file.Path` interface.
>
> Click here to view code image
>
> ```
> abstract Path getPath(String first, String... more)
> ```
>
> This method is declared in the `java.nio.file.FileSystem` class.
>
> Click here to view code image
>
> ```
> static Path get(String first, String... more)
> static Path get(URI uri)
> ```
>
> These static methods are declared in the `java.nio.file.Paths` class.
>
> ```
> Path toPath()
> ```
>
> This method is declared in the `java.io.File` class.

Figure 21.1 *Creating* `Path` *Objects*

Creating `Path` Objects with the `Path.of()` Method

The simplest way to create a `Path` is to use the static factory method `Path.of(String first, String... more)`. It joins the `first` string with any strings in the variable arity parameter `more` to create a path string that is the basis of constructing a `Path` object. This method creates a `Path` object in accordance with the default file system.

For instance, the default file system can be queried for the platform-specific name separator for name elements in a path.

The three absolute paths below are equivalent on a Unix platform, as they create a `Path` object based on the same path string. The `nameSeparator` string is the platform-specific name separator obtained from the default file system. At (1) and (2) below, only the `first` string parameter is specified in the `Path.of()` method, and it is the basis for constructing a `Path` object. However, specifying the variable arity parameter where possible is recommended when a `Path` object is constructed as shown at (3), as joining of the name elements is implicitly done using the platform-specific name separator. The `equals()` methods simply checks for equality on the path string, not on any directory entry denoted by the `Path` objects.

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // Obtain the default file syst
String nameSeparator = dfs.getSeparator();     // The name separator for a pat

Path absPath1 = Path.of("/a/b/c");                         // (1) /a/b/c
Path absPath2 = Path.of(nameSeparator + "a" +              // (2) /a/b/c
                        nameSeparator + "b" +
                        nameSeparator + "c");
Path absPath3 = Path.of(nameSeparator, "a", "b", "c");     // (3) /a/b/c
System.out.println(absPath1.equals(absPath2) &&
                   absPath2.equals(absPath3));             // true
```

The two absolute paths below are equivalent on a Windows platform, as they create `Path` objects based on the same path string. Note that the backslash character must be escaped with a second backslash in a string. Otherwise, it will be interpreted as starting an escape sequence (§2.1, p. 38).

Click here to view code image

```
Path absPath4 = Path.of("C:\\a\\b\\c");                    // (4) C:\a\b\c
Path absPath5 = Path.of("C:", "a", "b", "c");              // (5) C:\a\b\c
```

The `Path` created below is a relative path, as no root component is specified in the arguments.

Click here to view code image

```
Path relPath1 = Path.of("c", "d");                         //  c/d
```

Often we need to create a `Path` object to denote the current directory. This can be done via a *system property* named `"user.dir"` that can be looked up, as shown at (1) below, and its value used to construct a `Path` object, as shown at (2). The path string of the current directory can be used to create paths relative to the current directory, as shown at (3).

Click here to view code image

```
String pathOfCurrDir = System.getProperty("user.dir");  // (1)
Path currDir = Path.of(pathOfCurrDir);                   // (2)
Path relPath = Path.of(pathOfCurrDir, "d");              // (3) <curr-dir-path>
```

Creating `Path` Objects with the `Paths.get()` Method

The `Paths` utility class provides the `get(String first, String... more)` static factory method to construct `Path` objects. In fact, this method invokes the `Path.of(String first, String... more)` convenience method to obtain a `Path`.

Click here to view code image

```
Path absPath7 = Paths.get(nameSeparator, "a", "b", "c");
Path relPath3 = Paths.get("c", "d");
```

Creating `Path` Objects Using the Default File System

We have seen how to obtain the default file system that is accessible to the JVM:

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

The default file system provides the `getPath(String first, String... more)` method to construct `Path` objects. In fact, the `Path.of(String first, String... more)` method is a convenience method that invokes the `FileSystem.getPath()` method to obtain a `Path` object.

Click here to view code image

```
Path absPath6 = dfs.getPath(nameSeparator, "a", "b", "c");
Path relPath2 = dfs.getPath("c", "d");
```

Interoperability with the `java.io.File` Legacy Class

An object of the legacy class `java.io.File` can be used to query the file system for information about a file or a directory. The class also provides methods to create, rename, and delete directory entries in the file system. Although there is an overlap of functionality between the `Path` interface and the `File` class, the `Path` interface is recommended over the `File` class for new code. The interoperability between a `File` object and a `Path` object allows the limitations of the legacy class `java.io.File` to be addressed.

> Click here to view code image
>
> ```
> File(String pathname)
> Path toPath()
> ```
>
> This constructor and this method of the `java.io.File` class can be used to create a `File` object from a pathname and to convert a `File` object to a `Path` object, respectively.
>
> ```
> default File toFile()
> ```
>
> This method of the `java.nio.file.Path` interface can be used to convert a `Path` object to a `File` object.

The code below illustrates the round trip between a `File` object and a `Path` object:

Click here to view code image

```
File file = new File(File.separator + "a" +
                     File.separator + "b" +
                     File.separator + "c");          // /a/b/c

// File --> Path, using the java.io.File.toPath() instance method
Path fileToPath = file.toPath();                     // /a/b/c

// Path --> File, using the java.nio.file.Path.toFile() default method.
File pathToFile = fileToPath.toFile();               // /a/b/c
```

Interoperability with the `java.net.URI` Class

A URI consists of a string that identifies a resource, which can be a local resource or a remote resource. Among other pertinent information, a URI specifies a *scheme* (e.g., `file`, `ftp`, `http`, and `https`) that indicates what protocol to use to handle the resource. The examples of URIs below show two *schema*: `file` and `http`. We will not elaborate on the syntax of URIs, or different schema, as it is beyond the scope of this book.

Click here to view code image

```
file:///a/b/c/d                        // Scheme: file, to access a local file.
http://www.whatever.com                // Scheme: http, to access a remote websi
```

> Click here to view code image

> ```
> URI(String str) throws URISyntaxException
> static URI create(String str)
> ```
>
> This constructor and this static method in the `java.net.URI` class create a `URL` object based on the specified string. The second method is preferred when it is known that the URI string is well formed.

Click here to view code image

```
// Create a URI object, using the URL.create(String str) static factory method
URI uri1 = URI.create("file:///a/b/c/d");   // Local file.
```

The following methods can be used to convert a `URI` object to a `Path` object. The `Paths.get(URI uri)` static factory method actually invokes the `Path.of(URI uri)` static factory method.

Click here to view code image

```
// URI --> Path, using the Path.of(URI uri) static factory method.
Path uriToPath1 = Path.of(uri1);        // /a/b/c/d

// URI --> Path, using the Paths.get(URI uri) static factory method.
Path uriToPath2 = Paths.get(uri1);      // /a/b/c/d
```

The following method in the `Path` interface can be used to convert a `Path` object to a `URI` object:

Click here to view code image

```
// Path --> URI, using the Path.toUri() instance method.
URI pathToUri = uriToPath1.toUri();     // file:///a/b/c/d
```

Interoperability between a `Path` object and a `URI` object allows an application to leverage both the NIO.2 API and the network API.

21.3 Working with `Path` Objects

The `java.nio.file.Path` interface provides a myriad of methods to query and manipulate `Path` objects. In this section, selected methods from the `Path` interface are presented for working on `Path` objects. A majority of these methods perform syntactic operations on the path string contained in a `Path` object. As `Path` objects are immutable, the methods return a new `Path` object, making it possible to use this distinctive style of method chaining. There is also no requirement that the path string in a `Path` object must refer to an existing resource. Very few methods enforce this requirement, and if they do, they will throw a checked `IOException` if that it is not the case.

Querying `Path` Objects

The following methods of the `java.nio.file.Path` interface can be used for querying a `Path` object for its various properties. The names of most methods reflect their operation. The description in the API of these methods and Example 21.1 below should aid in understanding their functionality.

> `String toString()`
>
> Returns the text representation of this path.
>
> `boolean isAbsolute()`
>
> Determines whether this path is an absolute path or not.
>
> `FileSystem getFileSystem()`
>
> Returns the file system that created this object. Each `Path` object is created with respect to a file system.
>
> `Path getFileName()`
>
> Returns the name of the directory entry denoted by this path as a `Path` object— that is, the *last* name element in this path which denotes either a file or a directory.
>
> `Path getParent()`
>
> Returns the parent path, or null if this path does not have a parent—that is, logically going up one level in the directory hierarchy from the current position given by this path. It returns `null` if this `Path` does not have a parent—for example, if the path denotes the root component or it is a relative path that comprises a single name element, as there is no parent in these cases.
>
> `Path getRoot()`
>
> Returns the root component of this path as a `Path` object, or `null` if this path does not have a root component.
>
> `int getNameCount()`
>
> Returns the number of name elements in the path. It returns the value *n*, where `n-1` is the index of the last name element and the first name element has index `0`. Note that the root

component is not included in the count of the name elements.

```
Path getName(int index)
```

Returns a name element of this path as a `Path` object, where the name element closest to the root component has index `0`.

Click here to view code image

```
Path subpath(int beginIndex, int endIndex)
```

Returns a relative `Path` that is a subsequence of the name elements of this path, where `beginIndex` is inclusive but `endIndex` is exclusive (i.e., name elements in the range `[beginIndex, endIndex)`). Illegal indices will result in an `Illegal-ArgumentException`.

Click here to view code image

```
boolean startsWith(Path other)
default boolean startsWith(String other)
```

Determine whether this path starts with either the given path or a `Path` object constructed from the given `other` string. The methods can be used for conditional handing of paths.

Click here to view code image

```
boolean endsWith(Path other)
default boolean endsWith(String other)
```

Determine whether this path ends with either the given path or a `Path` object constructed from the given `other` string. These methods can be used for conditional handing of paths.

Example 21.1 illustrates methods for querying a `Path` object. An absolute path is created at (1). The output from the program shows the result of executing methods for text representation at (2), determining whether a path is absolute at (3), which file system created the path at (4), accessing name elements at (5), and testing the prefix and the suffix of a path at (6).

Example 21.1 *Querying* `Path` *Objects*

Click here to view code image

```
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
```

```
public class QueryingPaths {

  public static void main(String[] args) {
    FileSystem dfs = FileSystems.getDefault();     // The default file system
    String nameSeparator = dfs.getSeparator();     // The name separator

    Path absPath = Path.of(nameSeparator, "a", "b", "c");                   // (1

    System.out.printf("toString(): %s%n",   absPath);                       // (2
    System.out.printf("isAbsolute(): %s%n", absPath.isAbsolute());          // (3
    System.out.printf("getFileSystem(): %s%n",
                       absPath.getFileSystem().getClass().getName());       // (4

    System.out.println("\n***Accessing Name Elements:");                    // (5
    System.out.printf("getFileName(): %s%n",  absPath.getFileName());
    System.out.printf("getParent(): %s%n",    absPath.getParent());
    System.out.printf("getRoot(): %s%n",      absPath.getRoot());
    System.out.printf("getNameCount(): %d%n", absPath.getNameCount());

    List<Path> pl = new ArrayList<>();
    absPath.forEach(p -> pl.add(p));
    System.out.printf("List of name elements: %s%n",  pl);

    System.out.printf("getName(0): %s%n",     absPath.getName(0));
    System.out.printf("subpath(0,2): %s%n",   absPath.subpath(0,2));

    System.out.println("\n***Path Prefix and Suffix:");                     // (6
    System.out.printf("startsWith(\"%s\"): %s%n",
                      nameSeparator + "a",
                      absPath.startsWith(nameSeparator + "a"));
    System.out.printf("endsWith(\"b/c\"): %s%n",
                      absPath.endsWith("b/c"));
  }
}
```

Possible output from the program:

Click here to view code image

```
toString(): /a/b/c
isAbsolute(): true
getFileSystem(): sun.nio.fs.MacOSXFileSystem

***Accessing Name Elements:
getFileName(): c
getParent(): /a/b
getRoot(): /
getNameCount(): 3
List of name elements: [a, b, c]

getName(0): a
subpath(0,2): a/b

***Path Prefix and Suffix:
startsWith("/a"): true
endsWith("b/c"): true
```

Converting `Path` Objects

The `Path` interface provides many methods for converting paths. A majority of these methods manipulate the path strings syntactically and do not assume that the path strings of the `Path` objects denote actual directory entries in the file system.

> ```
> Path toAbsolutePath()
> ```
>
> Returns a `Path` object representing the absolute path of this `Path` object. If this `Path` object represents a relative path, an absolute path is constructed by appending it to the absolute path of the current directory. If this `Path` is an absolute path, it just returns this `Path` object.
>
> ```
> Path normalize()
> ```
>
> Returns a `Path` object that is created from this `Path` object with redundant name elements eliminated. This typically involves eliminating the `"."` and `"dir/.."` strings in this `Path` object, as these do not change the hierarchy of a path. The method applies the elimination procedure repeatedly until all such redundancies are eliminated.
>
> Click here to view code image
>
> ```
> Path resolve(Path other)
> default Path resolve(String other)
> ```
>
> These method resolve the given `Path` object (either specified or created from the specified string) against this `Path` object.
>
> If the `other Path` object represents an absolute path, the `other Path` object is returned as the result, regardless of whether this `Path` object represents an absolute or a relative path. Otherwise, the method creates a result `Path` object by *joining* the `other Path` object to this `Path` object.
>
> Click here to view code image
>
> ```
> default Path resolveSibling(Path other)
> default Path resolveSibling(String other)
> ```
>
> Resolve the given `Path` object (either specified or created from the specified string) against this `Path` object's *parent* path. That is, they resolve the given `Path` object by calling the `resolve(other)` method on this `Path` object's parent path.
>
> ```
> Path relativize(Path other)
> ```
>
> Constructs a relative `Path` object between this `Path` object and the given `other Path` object. The constructed relative `Path` object when resolved against this `Path` object should yield a path that represents the same directory entry as the given `other Path` object.

`A relative Path object can only be constructed when this Path object and the given Path object` *`both`* `represent either absolute paths or relative paths.`

If this path is `/w/x` and the given path is `/w/x/y/z`, the resulting relative path is `y/z`. That is, the given path `/w/x/y/z` and the resulting relative path `y/z` represent the same directory entry.

For two `Path` objects that are equal, the empty path is returned.

The `relativize()` method is the inverse of the `resolve()` method.

Click here to view code image

```
Path toRealPath(LinkOption... options) throws IOException
```

Returns a `Path` object that represents the *real* path of an existing directory entry. Note that this method throws an `IOException` if the path does not exist in the file system. It converts the path to an absolute path and removes any redundant elements, and does not follow symbolic links if the enum constant `Link-Option.NOFOLLOW_LINKS` is specified (p. 1301).

Converting a Path to an Absolute Path

The method `toAbsolutePath()` of the `Path` interface converts a path to an absolute path. Table 21.2 illustrates how this method works for `Path` objects declared in the second column. A print statement that calls the method on each path declaration, analogous to the one below, creates the text representation of the resulting absolute path shown in the rightmost column. The current directory has the absolute path `/a/b`.

Click here to view code image

```
System.out.println(absPath1.toAbsolutePath()); // (1) /a
```

In Table 21.2, (1) and (2) show that if the `Path` object already represents an absolute path, the same `Path` object is returned. If the `Path` object represents a relative path, as shown at (3), (4), and (5), the resulting absolute path is created by appending the relative path to the absolute path of the current directory. The path need not exist in the file system, and the method does not attempt to clean up the path string of the resulting `Path` object—in contrast to the `normalize()` method (p. 1299).

Table 21.2 *Converting to an Absolute Path*

Absolute path of the current directory		Text representation of the absolute path, returned by the `toAbsolutePath()` method
(1)	`Path absPath1 = Path.of("/a");`	`/a`
(2)	`Path absPath2 = Path.of("/a/b/c");`	`/a/b/c`
(3)	`Path relPath1 = Path.of("d");`	`/a/b/d`
(4)	`Path relPath2 = Path.of("../f");`	`/a/b/../f`
(5)	`Path relPath3 = Path.of("./../g");`	`/a/b/./../g`

Normalizing a Path

The `normalize()` method of the `Path` interface removes redundancies in a `Path` object. For example, the current directory designator `"."` is redundant in a `Path` object, as it does not add any new level to the path hierarchy. Also the string `"`*`dir`*`/.."` in a path is redundant, as it implies going one level down in the path hierarchy and then one level up again—not changing the hierarchy represented by the path.

Table 21.3 illustrates how the `normalize()` method converts the `Path` objects declared in the second column. A print statement that calls the method on each path declaration, analogous to the one below, creates the text representation of the resulting path shown in the rightmost column.

Click here to view code image

```
System.out.println(path1.normalize());          // (1) a/b/c
```

The numbers below refer to the rows in Table 21.3.

(1) All occurrences of the current directory designator `"."` are redundant and are eliminated from the path.

(2) The occurrences of the redundant string `"a/.."` are eliminated from the path.

(3) The occurrences of the parent directory designator `".."` are significant in this case, as each occurrence implies traversing one level up the path hierarchy.

(4) All occurrences of the current directory designator `"."` are eliminated from the path, but occurrences of the parent directory designator `".."` are significant.

(5) The occurrence of the redundant current directory designator `"."` is eliminated from the path, resulting in

an empty path whose text representation is the empty string.

Because of redundancies in a path, comparison on paths is best performed on normalized paths (p. 1303).

Table 21.3 *Normalizing Paths*

Path	**Text representation of the path returned by the `normalize()` method**
`Path path1 = Path.of("./a/./b/c/.");`	(1) `a/b/c`
`Path path2 = Path.of("a/../a/../b");`	(2) `b`
`Path path3 = Path.of("../../d");`	(3) `../../d`
`Path path4 = Path.of("./../.././.");`	(4) `../..`
`Path path5 = Path.of(".");`	(5) *empty string*

Resolving Two Paths

Table 21.4 illustrates how the `resolve()` method of the `Path` interface performs resolution between two paths. The four combinations of mixing absolute and relative paths when calling the `resolve()` method are represented by the rows (R1 and R2) and the columns (C1 and C2) in Table 21.4. The results shown are obtained by executing a print statement that calls the `resolve()` method, analogous to the one below, for each combination.

Click here to view code image

```
System.out.println(absPath1.resolve(absPath2));      // (R1, C1)
```

If the given path is an absolute path, it is returned as the result, as in column C1, regardless of whether the path on which the method is invoked is an absolute or a relative path. Otherwise, the method creates a result path by appending the given path to the path on which the method is invoked, as in column C2.

In the special case when the given path is an empty path, the method returns the path on which it was invoked:

Click here to view code image

```
Path anyPath = Path.of("/a/n/y");
Path emptyPath = Path.of("");
System.out.println(anyPath.resolve(emptyPath));       // /a/n/y
```

Note that the paths need not exist to use this method, and the resulting path after resolution is not normalized.

Table 21.4 *Resolving Paths*

`p1.resolve(p2`	**C1**	**C2**	**`elPath1`, and**

	where *p2* can be **`absPath2`** or **`relPath2`**	
	`Path absPath2 = Path.of("/c");` `Path relPath2 = Path.of("e/f");`	
R1 `Path absPath1 = Path.of("/a/b");`	`/a/b` `../c` `./e/f`	
R2 `Path relPath1 = Path.of("d");`	`d/` `../e/f`	

Constructing the Relative Path between Two Paths

Table 21.5 illustrates how the `relativize()` method of the `Path` interface constructs a relative path between this path and the given path, so that the resulting relative path denotes the same directory entry as the given path.

The four combinations of mixing absolute and relative paths when calling the `relativize()` method are represented by the rows (R1 and R2) and the columns (C1 and C2) in Table 21.5. The results shown are obtained by executing a print statement that calls the `relativize()` method, analogous to the one below, for each combination.

Click here to view code image

```
System.out.println(absPath1.revitalize(absPath2));     // (R1, C1)
```

For the case (R1, C1) where both paths are absolute paths in Table 21.5, the resulting relative path is constructed relative to the root of the directory hierarchy. The relative path between the absolute path `/a/b` and the given absolute path `/c` is `../../c`, indicating traversing two levels up from the path `/a/b` to the root and joining with the given path `/c`. Both the resulting relative path `../../c` and the given path `/c` denote the same directory entry.

For the case (R2, C2) where both paths are relative paths in Table 21.5, the resulting relative path is constructed relative to the *current directory*. The relative path between the relative path `d` and the given relative path `e/f` is `../e/f`, indicating traversing one level up from the path `d` to the current directory and joining with the given path `e/f`. Both the resulting relative path `../e/f` and the given path `e/f` denote the same directory entry.

From Table 21.5, we see that an `IllegalArgumentException` is thrown when both paths are neither absolute paths nor relative paths, as it is not possible to create a relative path between paths that do not satisfy this criteria.

The code below shows the relationship between the `relativize()` and the `resolve()` methods:

Click here to view code image

```
Path p = Path.of("/a/b");
Path other = Path.of("/a/b/c/d");
Path q = p.relativize(other);                                // c/d
System.out.println(p.relativize(p.resolve(q)).equals(q));    // true
System.out.println(p.resolve(q).equals(other));              // true
```

Note that the paths need not exist to use this method, as it operates syntactically on the path strings.

Table 21.5 *Constructing a Relative Path between Two Paths*

`p1.relativize(p2)` where *p1* can be `absPath1` or `relPath1`, and where *p2* can be `absPath2` or `relPath2`	C1 `Path absPath2 = Path.of("/c/e");`	C2 `Path relPath2 = Path.of("c/f");`
R1 `Path absPath1 = Path.of("/a/b");`	`../../c/e`	`IllegalArgumentException`
R2 `Path relPath1 = Path.of("d");`	`IllegalArgumentException`	`../c/f`

Link Option

The enum type `LinkOption` defines how symbolic links should be handled by a file operation. This enum type defines only one constant, shown in Table 21.6. If the constant `NOFOLLOW_LINKS` is specified in a method call, symbolic links are not followed by the method.

The enum type `LinkOption` implements both the `CopyOption` interface (p. 1308) and the `OpenOption` interface (p. 1314). Many file operations declare a variable arity parameter of one of these interface types or just the enum type `LinkOption`, making it possible to configure their operation.

Table 21.6 *Link Option*

Enum **`java.nio.file.LinkOption`** implements the **`java.nio.file.CopyOption`** and the **`java.nio.file.OpenOption`** interfaces	Description
`NOFOLLOW_LINKS`	Do not follow symbolic links.

Converting a Path to a Real Path

The `toRealPath()` method of the `Path` interface converts this path to an absolute path that denotes the same directory entry as this path. The name elements in the path must represent actual directories in the file system or the method throws an `IOException`. It accepts a variable arity parameter of the enum type `java.nio.file.LinkOption` (Table 21.6). If the variable arity parameter is not specified, symbolic links are followed to their final target.

The `toRealPath()` method performs several operations to construct the real path:

- If this path is a relative path, it is first converted to an absolute path.
- Any redundant name elements are removed to create a new path. In other words, it normalizes the result path.
- If `LinkOption.NOFOLLOW_LINKS` is specified, any symbolic links are not followed.

Table 21.7 show the results of calling the `toRealPath()` method on selected paths. Since the method throws an `IOException`, it should typically be called in a `try-catch` construct.

Click here to view code image

```
try {
  Path somePath = Path.of("some/path");
  Path realPath = somePath.toRealPath(LinkOption.NOFOLLOW_LINKS);
  System.out.println(realPath);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Table 21.7 *Converting to a Real Path*

Text representation of the path	Path returned by the `toRealPath()` method
Current directory is `/book/chap01`. The symbolic link `alias_appendixA` has the target `/book/appendixA`.	
(1) `Path currDir = Path.of(".");`	`/book/chap01`
(2) `Path parentDir = Path.of("..");`	`/book`
(3) `Path [illegible] = Path.of("./examples/.././examples/D.java");`	`/book/chap01/examples/D.java`
(4) `Path path4 = Path.of("alias_appendixA");`	`path4.toRealPath()` returns `/book/appendixA`. `path4.toRealPath(LinkOption.NOFOLLOW_LINKS)` returns `/book/chap01/alias_appendixA`.

Comparing `Path` Objects

The methods in the `Path` interface for comparing two paths only consult the path strings, and do not access the file system or require that the paths exist in the file system.

Click here to view code image

`boolean equals(Object other)`

Determines whether this path is equal to the given object by comparing their path strings. It does *not* eliminate any redundancies from the paths before testing for equality. See also the `Files.isSameFile()` method (p. 1306).

`int compareTo(Path other)`

A `Path` object implements the `Comparable<Path>` interface. This method compares two path strings lexicographically according to the established contract of this method. It means paths can be compared, searched, and sorted.

The code below illustrates sorting paths according to their natural order. Comparison of the paths is based purely on comparison of their path strings, as is the equality comparison.

Click here to view code image

```
Path p1 = Path.of("/", "a", "b", "c", "d");
Path p2 = Path.of("/", "a", "b");
Path p3 = Path.of("/", "a", "b", "c");
Path p4 = Path.of("a", "b");

// Sorting paths according to natural order:
List<Path> sortedPaths = Stream.of(p1, p2, p3, p4)
                               .sorted()
                               .toList();
System.out.println(sortedPaths);
// [/a/b, /a/b/c, /a/b/c/d, a/b]

// Comparing for lexicographical equality:
System.out.println(p2);                          // Absolute path: /a/b
System.out.println(p3.subpath(0, 2));            // Relative path: a/b
System.out.println(p2.equals(p3.subpath(0, 2))); // false
```

21.4 Operations on Directory Entries

The static methods of the `Files` class interact with the file system to access directory entries in the file system. The methods make heavy use of `Path` objects that denote directory entries.

Characteristics of Methods in the `Files` Class

It is worth noting certain aspects of the static methods in the `Files` class, as this will aid in using and understanding the operations they perform.

Handling System Resources

The NIO.2 API uses many resources, such as files and streams, that should be closed after use to avoid any degradation of performance due to lack of system resources. As these resources implement the `java.io.Closeable` interface, they are best handled in a `try`-with-resources statement that guarantees to close them after its execution.

Handling Exceptions

Errors are bound to occur when interacting with the file system. Numerous errors can occur, and among the most common errors are the following:

- A required file or directory does not exist in the file system.
- Permissions to access a file are incorrect.

A majority of the static methods in the `Files` class throw a `java.io.IOException` that acts as the catchall for various I/O errors. Its subclass, `java.nio.file.NoSuchFile-Exception`, unequivocally makes clear the cause of the exception. As these exceptions are checked exceptions, code using these methods should diligently handle these exceptions with either a `try-catch-finally` construct or a `throws` clause.

For brevity, the exception handling may be omitted in some code presented in this chapter.

Handling Symbolic Links

The methods of the `Files` class are savvy with regard to symbolic links. Following of symbolic links is typically indicated by specifying or omitting the constant `LinkOption.NOFOLLOW_LINKS` for the variable arity parameter of the method (see below).

Specifying Variable Arity Parameters

Many static methods in the `Files` class have a variable arity parameter. Such a parameter allows zero or more options to customize the operation implemented by the method. For example, the following method takes into consideration symbolic links depending on whether or not the variable arity parameter `options` is specified:

Click here to view code image

```
// Method header:
static boolean exists(Path path, LinkOption... options)  // Variable arity par

// Method calls:
Path path = Path.of("alias");
boolean result1 = Files.exists(path);            // Follow symbolic links.
boolean result2 = Files.exists(path,
                   LinkOption.NOFOLLOW_LINKS);  // Do not follow symbolic lin
```

Executing Atomic Operations

Certain file operations can be performed as *atomic operations*. Such an operation guarantees the integrity of any resource it uses. It runs independently and cannot be interrupted by other operations that might be running concurrently. See the *atomic move* operation (p. 1305).

Determining the Existence of a Directory Entry

The following static methods of the `Files` class test the existence or nonexistence of a directory entry denoted by a path.

> Click here to view code image
>
> ```
> static boolean exists(Path path, LinkOption... options)
> static boolean notExists(Path path, LinkOption... options)
> ```
>
> The first method tests whether a directory entry exists. The second method tests whether the directory entry denoted by this path does not exist.
>
> These methods normalize the path, and do not follow symbolic links if the enum constant `LinkOption.NOFOLLOW_LINKS` is specified for the `options` variable arity parameter.
>
> These methods are *not* complements of each other. Both return `false` if they are not able to determine whether a directory entry exists or not. This can occur due to lack of access permissions.
>
> Note that the result returned by these methods is immediately outdated. The outcome of subsequent access of the directory entry is unpredictable, as concurrently running threads might change the conditions after the method returns.

Although a `Path` object can represent a hypothetical path, ultimately the existence of the directory entry it denotes must be verified by interacting with the file system.

Given the following directory hierarchy, the code below demonstrates what result is returned by the `exists()` and `notExists()` methods of the `Files` class.

Click here to view code image

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
System.out.println(Files.exists(path1));                                // true
System.out.println(Files.notExists(path1));                             // false

Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java
System.out.println(Files.exists(path2));                                // true
System.out.println(Files.notExists(path2));                             // false

Path path3 = Path.of("project", "readme.txt");
System.out.println(Files.exists(path3));                                // false
System.out.println(Files.notExists(path3));                             // true
```

Given that the path `./project/manifest_link` is a symbolic link to the path `./project/ src/manifest.txt`, the code below demonstrates following symbolic links in the `exists()` method.

Click here to view code image

```
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

boolean result4 = Files.exists(target);                                 // (1)
boolean result5 = Files.exists(symbLink);                               // (2)
boolean result6 = Files.exists(symbLink, LinkOption.NOFOLLOW_LINKS);    // (3)

System.out.println("target: " + result4);                               // (1a) t
System.out.println("symbLink->target: " + result5);                     // (2a) t
System.out.println("symbLink_NOFOLLOW_LINKS: " + result6);              // (3a) t
```

Note that (1) and (2) above are equivalent. The existence of the target is tested at (2) as the symbolic link is followed by default. Whereas at (3), the existence of the symbolic link itself is tested, since the enum constant `LinkOption.NOFOLLOW_LINKS` is specified.

Uniqueness of a Directory Entry

The method `isSameFile()` in the `Files` class can be used to check whether two paths denote the *same* directory entry. It does not take into consideration other aspects of the directory entry, like its file name or contents.

Click here to view code image

```
static boolean isSameFile(Path path1, Path path2) throws IOException
```

> Determines whether the two paths denote the same directory entry. If the paths are equal, it returns `true` and does not check whether the paths exist.
>
> This method normalizes the paths and follows symbolic links.
>
> It implements an *equivalence relation* (which is *reflexive*, *symmetric*, and *transitive*) for non-`null` paths, if the file system and the directory entries do not change.

The numbers below refer to corresponding lines in the code to illustrate the workings of the `isSameFile()` method:

(1) Paths are always normalized, as in the case of `path2`.

(2) Symbolic links are always followed, as in the case of `symbLink`.

(3) Only paths are compared. Paths passed to the method are not equal.

(4) Equal paths return `true`, and their existence is not checked. The path `./Main.java` does not exist.

(5) The paths must exist in the file system, if they are not equal. The path `./Main.java` does not exist, resulting in a `java.nio.file.NoSuchFileException`.

Click here to view code image

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java

Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

System.out.println(Files.isSameFile(path1, path2));            // (1) true
System.out.println(Files.isSameFile(symbLink, target));        // (2) true
System.out.println(Files.isSameFile(path1, target));           // (3) false
System.out.println(Files.isSameFile(Path.of("Main.java"),
                                    Path.of("Main.java")));    // (4) true
System.out.println(Files.isSameFile(path1,
                           Path.of("Main.java")));  // (5) NoSuchFileException
```

Deleting Directory Entries

The methods `delete()` and `deleteIfExists()` in the `Files` class can be used for deleting directory entries.

> Click here to view code image
>
> ```
> static void delete(Path path) throws IOException
> ```

> ```
> static boolean deleteIfExists(Path path) throws IOException
> ```
>
> Delete a directory entry denoted by the `path`.
>
> If the path does not exist, the first method throws a `NoSuchFileException`, but the second method does not.
>
> Deleting a symbolic link only deletes the link, and not the target of the link.
>
> To delete a directory, it must be empty or a `java.nio.file.DirectoryNotEmpty-Exception` is thrown.

Consider the following paths that exist:

Click here to view code image

```
Path projDir  = Path.of("project");
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");
```

The `delete()` method throws a `NoSuchFileException`, if the path does not exist.

Click here to view code image

```
Files.delete(symbLink);                 // Exists. Link deleted, not target.
Files.delete(Path.of("Main.java"));     // Does not exist: NoSuchFileException
```

The `deleteIfExists()` method does not throw a `NoSuchFileException`. It indicates the result by the `boolean` return value.

Click here to view code image

```
System.out.println(Files.deleteIfExists(target));                  // Exists.
                                                                   //  Deleted: t
System.out.println(Files.deleteIfExists(Path.of("Main.java"))); // Does not
                                                                   //   exist: fa
```

In order to delete a directory, it must be empty. The directory `./project` is not empty. Both methods throw a `DirectoryNotEmptyException`.

Click here to view code image

```
Files.delete(projDir);                             // DirectoryNotEmptyExcepti
System.out.println(Files.deleteIfExists(projDir)); // DirectoryNotEmptyExcepti
```

Also keep in mind that deleting a directory entry might not be possible, if another program is using it.

Copy Options

The enum types `LinkOption` (Table 21.6) and `StandardCopyOption` (Table 21.8) implement the `CopyOption` interface. These options can be used to configure copying and moving of directory entries. A variable arity parameter of type `CopyOption...` is declared by the `copy()` and `move()` methods of the `Files` class that support specific constants of the `LinkOption` and `StandardCopyOption` enum types.

Table 21.8 *Standard Copy Options*

`java.nio.file.Standard` **`java.nio.file.CopyOption`** **interface**	**Description**
`REPLACE_EXISTING`	Replace a file if it exists.
`COPY_ATTRIBUTES`	Copy file attributes to the new file (p. 1321).
`ATOMIC_MOVE`	Move the file as an atomic file system operation—that is, an operation that is either performed uninterrupted in its entirety, or it fails.

Copying Directory Entries

The overloaded `copy()` methods of the `Files` class implement copying contents of files. Two of the `copy()` methods can be configured by specifying *copy options*.

> Click here to view code image
>
> ```
> static Path copy(Path source, Path destination, CopyOption... options)
>                 throws IOException
> ```
>
> Copies a `source` directory entry to the `destination` directory entry. It returns the path to the `destination` directory entry. The default behavior is outlined below, but can be configured by copy options:
>
> - If `destination` already exists or is a symbolic link, copying *fails*.
> - • If `source` and `destination` are the same, the method completes without any copying.
> - If `source` is a symbolic link, the target of the link is copied.
> - If `source` is a directory, just an *empty* `destination` directory is created.
>
> The following copy options can be specified to configure the

default copying behavior:

- `StandardCopyOption.REPLACE_EXISTING`: If the `destination` exists, this option indicates to replace the `destination` if it is a file or an empty directory. If the `destination` exists and is a symbolic link, it indicates to replace the symbolic link and not its target.
- `StandardCopyOption.COPY_ATTRIBUTES`: This option indicates to copy the file attributes of the `source` to the `destination`. However, copying of attributes is platform dependent.
- `LinkOption.NOFOLLOW_LINKS`: This option indicates not to follow symbolic links. If the `source` is a symbolic link, then the symbolic link is copied and not its target.

Click here to view code image

```
static long copy(InputStream in, Path destination, CopyOption... options)
                throws IOException
```

Copies all bytes from an input stream to a `destination` path, and returns the number of bytes copied. The input stream will be at the end of the stream after copying, but may not be because of I/O errors.

By default, if the `destination` path already exists or is a symbolic link, copying *fails*.

It can be configured by the following copy option:

- `StandardCopyOption.REPLACE_EXISTING`: If the `destination` path exists and is a file or an empty directory, this option indicates to replace the `destination`. If the `destination` exists and is a symbolic link, this option indicates to replace the symbolic link and not its target.

Click here to view code image

```
static long copy(Path source, OutputStream output) throws IOException
```

Copies all bytes from the `source` to the `output` stream, and returns the number of bytes copied. It may be necessary to flush the `output` stream.

> Note that this `copy()` method cannot be configured.
>
> The output stream may be in an inconsistent state because of I/O errors.

The first thing to keep in mind is that the copy methods do *not* create the intermediate directories that are in the destination path. Given the destination path `./project/archive/destFile`, the parent path `./project/archive` must exist. Otherwise, a `NoSuchFileException` is thrown.

Note also that copying fails if the destination path already exists—a `FileAlreadyExistsException` is thrown, unless the method is qualified by the enum constant `StandardCopyOption.REPLACE_EXISTING`.

Copy and Replace Directory Entries

Consider copying the source file:

```
./project/src/pkg/Main.java
```

to the destination file:

Click here to view code image

```
./project/archive/src/pkg/Main.java
```

The numbers below refer to corresponding lines in the code below to illustrate copying of files:

(1) Creates the `Path` object that denotes the source file.

(2) Creates the `Path` object for the *parent* path of the destination file. This parent path must exist.

(3) Resolves the parent path with the source pathname, so that if the destination file is to have the same name as the source file, its `Path` object can be constructed from the parent path and the file name of the source file.

Items (4) through (6) illustrate the scenario when calling the `copy()` method successively.

(4) Creates the destination file and copies the contents of the source file to the destination file.

(5) Overwrites the contents of the destination file with the contents of the source file.

(6) Fails, as the destination file already exists.

Click here to view code image

```
Path source = Path.of("project", "src", "pkg", "Main.java");                //
Path parentDestinationPath = Path.of("project", "archive", "src", "pkg");   //
Path destination = parentDestinationPath.resolve(source.getFileName());     //

Files.copy(source, destination);      // (4) OK. Destination file does not exi
Files.copy(source, destination,       // (5) OK. Destination file replaced.
           StandardCopyOption.REPLACE_EXISTING);
Files.copy(source, destination);      // (6) FileAlreadyExistsException
```

The `copy()` method does *not* copy the entries in a source directory to a destination directory. The code below attempts to copy the source directory:

```
./project/src
```

to the destination directory:

```
./project/backup
```

Possible outcomes of the following copying operation can be any of the bulleted options listed below:

Click here to view code image

```
Path srcDir  = Path.of("project", "src");
Path destDir = Path.of("project", "backup");
Files.copy(srcDir, destDir, StandardCopyOption.REPLACE_EXISTING);
```

- If an entry named `backup` does not exist in the `project` directory, an empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is an empty directory, a new empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is a file, the file is deleted and an empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is a non-empty directory, the copying operation fails with a `DirectoryNotEmptyException`.

Another special case to consider is copying a file to a directory. Consider the scenario when copying the source file:

```
./project/src/pkg/Main.java
```

to the destination directory:

```
./project/classes
```

The code and possible outcomes are outlined below.

Click here to view code image

```
Path srcFile = Path.of("project", "src", "pkg", "Main.java");
Path destDir = Path.of("project", "classes");
Files.copy(srcFile, destDir, StandardCopyOption.REPLACE_EXISTING);
```

- If an entry named `classes` does not exist in the `project` directory, a file named `classes` is created and the contents of the source file are copied to the file `classes`.
- If an entry named `classes` exists in the `project` directory and it is a file, the file is deleted, a new file named `classes` is created, and the contents of the source file are copied to the file `classes`.
- If an entry named `classes` exists in the `project` directory and it is an empty directory, the directory is deleted, a file named `classes` is created, and the contents of the source file are copied to the file `classes`.
- If an entry named `classes` exists in the `project` directory and it is a non-empty directory, the copying operation fails with a `DirectoryNotEmptyException`.

Copy Files Using I/O Streams

The `Files` class provides methods to copy files using byte I/O streams from the `java.io` package (§20.2, p. 1234). Bytes can be copied from a source file to an `Input-Stream` and from an `OutputStream` to a destination file. We demonstrate copying where input streams and output streams are assigned to files. See also reading and writing files using paths (p. 1314).

The code below reads bytes from the input file `project/src/pkg/Util.java` using a `BufferedInputStream` and writes the bytes to the output file path `project/backup/ Util.java`.

Click here to view code image

```
String inputFileName  = "project/src/pkg/Util.java";
Path outputFilePath = Path.of("project", "backup", "Util.java");
try (var fis = new FileInputStream(inputFileName);
     var bis = new BufferedInputStream(fis)) {
  long bytesCopied = Files.copy(bis, outputFilePath,
                                StandardCopyOption.REPLACE_EXISTING);
```

```
    System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 10
  }
```

The code below copies bytes from the input file path `project/backup/Util.java` to the output file `project/archive/src/pkg/Util.java` using a `BufferedOutputStream`.

Click here to view code image

```
  Path inputFilePath  = Path.of("project", "backup", "Util.java");
  String outputFileName = "project/archive/src/pkg/Util.java";
  try (var fos = new FileOutputStream(outputFileName);
       var bos = new BufferedOutputStream(fos)) {
    long bytesCopied = Files.copy(inputFilePath, bos);
    System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 10
  }
```

The following statement can be used to print the contents of a file to the standard output:

Click here to view code image

```
  Files.copy(inputFilePath, System.out);     // Prints file content to standard
```

In general, any `InputStream` or `OutputStream` can be used in the respective `copy()` methods.

Moving and Renaming Directory Entries

The `move()` method of the `Files` class implements moving and renaming directory entries. The method can be configured by specifying *copy options*.

The `move()` method emulates the copying behavior of the `copy()` method. But in contrast to the `copy()` method, the `move()` method deletes the source if the operation succeeds. Both methods allow the destination to be overwritten, if the constant `StandardCopyOption.REPLACE_EXISTING` is specified.

Click here to view code image

```
static Path move(Path source, Path destination, CopyOption... options)
            throws IOException
```

Moves or renames the `source` to the `destination`. After moving, the `source` is deleted. The method returns the path to the `destination`. The default behavior is outlined below, but can be configured by copy options:

- If `destination` already exists, the move *fails*.

> - If `source` and `destination` are the same, the method has no effect.
> - If `source` is a symbolic link, the target of the link is moved.
> - If `source` is an empty directory, the empty directory is moved to the `destination`.
>
> The following copy options can be specified to configure the default moving behavior:
>
> - `StandardCopyOption.REPLACE_EXISTING`: If the `destination` exists, this option indicates to replace the `destination` if it is a file or an empty directory. If the `destination` exists and is a symbolic link, this option indicates to replace the symbolic link and not its target.
> - `StandardCopyOption.ATOMIC_MOVE`: The move is performed as an *atomic file system operation*. It is implementation specific whether the move is performed if the destination exists or whether an `IOException` is thrown. If the move cannot be performed, the method throws an `AtomicMoveNotSupportedException`.

Moving Directory Entries

The code below illustrates moving a file (`./project/src/manifest.txt`) to a new location (`./project/bkup/manifest.txt`). If the file exists at the new location, it is replaced by the source file.

Click here to view code image

```
Path srcFile  = Path.of("project", "src",  "manifest.txt");
Path destFile = Path.of("project", "bkup", "manifest.txt");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING);
```

We can move a directory *and* its hierarchy (`./project/bkup`) to a new location (`./project/ archive/backup`). The directory (and its contents) are moved to the new location and renamed (`backup`).

Click here to view code image

```
Path srcDir = Path.of("project", "bkup");
Path destDir = Path.of("project", "archive", "backup");  // Parent path exists
Files.move(srcDir, destDir);
```

Renaming Directory Entries

The `move()` method can be used to rename a directory entry. The code below illustrates renaming an existing file (`Util.java`). Its name is changed (`UX.java`), but not its contents.

Click here to view code image

```
Path oldFileName  = Path.of("project", "backup", "Util.java");
Path newFileName  = Path.of("project", "backup", "UX.java");
Files.move(oldFileName, newFileName);
```

Analogously, the code below illustrates renaming an existing directory (`backup`). Its name is changed (`bkup`), but not its hierarchy.

Click here to view code image

```
Path oldDirName  = Path.of("project", "backup");
Path newDirName  = Path.of("project", "bkup");
Files.move(oldDirName, newDirName);
```

Atomic Move

The enum constant `StandardCopyOption.ATOMIC_MOVE` can be specified in the `move()` method to indicate an *atomic move*—that is, an operation that is indivisible. It either completes in its entirety or it fails. The upshot of an atomic operation is that other threads will never see incomplete or partial results. An `AtomicMoveNotSupported-Exception` will be thrown if the file system does not support this feature.

In the following code, the file `Util.java` in the directory `./project/src/pkg` is moved in an atomic operation to its new location `./project/archive/src/pkg`.

Click here to view code image

```
Path srcFile = Path.of("project", "src", "pkg", "Util.java");
Path destFile = Path.of("project", "archive", "src", "pkg", "Util.java");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING,
                              StandardCopyOption.ATOMIC_MOVE);
```

21.5 Reading and Writing Files Using Paths

The `Files` class provides methods for reading and writing bytes and characters using I/O streams and `Path` objects.

Methods provided for reading and writing files typically close the file after use.

Open Options

The `java.nio.file.OpenOption` interface is implemented by objects that can be specified as options to configure how a file operation should open or create a file. Methods for writing to files can be configured for this purpose by specifying constants defined by the enum type `java.nio.file.StandardOpenOption` that implements the `OpenOption` interface.

Table 21.9 shows the options defined by constants of the `StandardOpenOption` enum type. Such options are specified as values for the variable arity parameter of type `OpenOption` in methods such as `newBufferedWriter()`, `write()`, `writeString()`, `newOutputStream()`, and `newInputStream()` of the `Files` class.

For write operations, if no options are supplied, it implies that the following options are present: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`—meaning the file is opened for writing, created if it does not exist, and truncated to size 0.

Table 21.9 *Selected Standard Open Options*

java.nio.file.Standard **java.nio.file.OpenOption interface**	**Description**
`READ`	Open the file for read access.
`WRITE`	Open the file for write access.
`APPEND`	If the file is opened for `WRITE` access, write bytes to the end of the file. That is, its previous content is not overwritten.
`TRUNCATE_EXISTING`	If the file already exists and it is opened for `WRITE` access, truncate its length to 0 so that bytes are written from the beginning of the file.
`CREATE`	Open the file if it exists; otherwise, create a new file.
`CREATE_NEW`	Fail if the file already exists; otherwise, create a new file.
`DELETE_ON_CLOSE`	Delete the file when the stream is closed. Typically used for temporary files.

Reading and Writing Character Data

The `Files` class provides methods for reading and writing *character data* to files. These methods can be categorized as follows:

- Methods that create character I/O streams (`BufferedReader`, `BufferedWriter`) chained to a `Path` object that denotes a file. The methods of the buffered reader and writer can then be used to read and write characters to the file, respectively.

- Methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object.

Reading and Writing Character Data Using Buffered I/O Streams

The `newBufferedReader()` and `newBufferedWriter()` methods of the `Files` class create buffered readers and writers, respectively, that are chained to a `Path` object denoting a file. Interoperability between character I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing character data to a file (§20.3, p. 1241).

Previously we have used constructors of the `BufferedReader` class (§20.3, p. 1251) and the `BufferedWriter` class (§20.3, p. 1250) in the `java.io` package to instantiate buffered readers and writers that are chained to a `Reader` or a `Writer`, respectively. Using the methods of the `Files` class is the recommended practice for creating buffered readers and writers when dealing with text files.

> Click here to view code image
>
> ```
> static BufferedReader newBufferedReader(Path path) throws IOException
> static BufferedReader newBufferedReader(Path path, Charset cs)
>                                                    throws IOException
> ```
>
> Opens the file denoted by the specified `path` for reading, and returns a `BufferedReader` of a default size to read text efficiently from the file, using either the UTF-8 charset or the specified charset to decode the bytes, respectively. Contrast these methods with the constructors of the `java.io.BufferReader` class (§20.3, p. 1251).
>
> Click here to view code image
>
> ```
>   static BufferedWriter newBufferedWriter(Path path, OpenOption... options)
>                                                      throws IOException
>   static BufferedWriter newBufferedWriter(Path path, Charset cs,
>                         OpenOption... options) throws IOException
> ```
>
> Opens or creates a file denoted by the specified `path` for writing, returning a `BufferedWriter` of a default size that can be used to write text efficiently to the file, using either the UTF-8 charset or the specified charset to encode the

> characters, respectively. See also constructors of the `java.io.BufferWriter` class (§20.3, p. 1250).

The code at (1) and at (3) in Example 21.2 illustrates writing lines to a text file using a buffered writer and reading lines from a text file using a buffered reader, respectively. The methods `newBufferedWriter()` and `newBufferedReader()` create the necessary buffered writer and reader at (2) and (4), respectively, whose methods are used to write and read the lines from the file.

Example 21.2 *Reading and Writing Text Files*

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ReadingWritingTextFiles {

  public static void main(String[] args) throws IOException {
    // List of strings:
    List<String> lines = List.of("Guess who got caught?", "Who?",
                                 "NullPointerException.",
                                 "Seriously?", "No. Finally.");
    // Text file:
    String filename = "project/linesOnly.txt";
    Path path = Path.of(filename);

    // Writing lines using buffered writer:                             (1)
    try (BufferedWriter writer =  Files.newBufferedWriter(path)) {   // (2)
      for(String str : lines) {
        writer.write(str);                  // Write a string.
        writer.newLine();                   // Terminate with a newline.
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    // Read lines using buffered reader:                                (3)
    lines = new ArrayList<>();
    try(BufferedReader reader= Files.newBufferedReader(path)) {      // (4)
      String line = null;
      while ((line = reader.readLine()) != null) {  // EOF when null is return
        lines.add(line);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    System.out.printf("Lines read from file \"%s\":%n%s%n", path, lines);


    // Write the list of strings in one operation:
    Files.write(path, lines);                                        // (5)

    // Write the joined lines in one operation:
    String joinedLines = String.join(System.lineSeparator(), lines);
```

```
      Files.writeString(path, joinedLines);                          // (6)

      // Read all contents into a String, including line separators:
      String allContent = Files.readString(path);                    // (7)
      System.out.printf("All lines read from file \"%s\":%n%s%n", path, allConte

      // Read all lines into a list of String:
      lines = Files.readAllLines(path);                              // (8)
      System.out.printf("List of lines read from file \"%s\":%n%s%n", path, line
    }
  }
```

Output from the program:

Click here to view code image

```
Lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
All lines read from file "project/linesOnly.txt":
Guess who got caught?
Who?
NullPointerException.
Seriously?
No. Finally.
List of lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
```

Reading and Writing Character Data Using Path Objects

The `Files` class provides methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object, without the need to specify an I/O stream. These methods also close the file when done.

Click here to view code image

```
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  OpenOption... options) throws IOException
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  Charset cs, OpenOption... options) throws IOException
```

Open or create a file denoted by the specified `path`, and writes text `lines` to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Click here to view code image

```
static Path writeString(Path path, CharSequence csq, OpenOption... options)
            throws IOException
static Path writeString(Path path, CharSequence csq, Charset cs,
                        OpenOption... options) throws IOException
```

Open or create a file denoted by the specified `path`, and writes characters in the `CharSequence csq` verbatim to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Click here to view code image

```
static String readString(Path path) throws IOException
static String readString(Path path, Charset cs) throws IOException
```

Read all content from a file denoted by the specified `path` into a string, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. The string returned will contain all characters, including line separators. These methods are not recommended for reading large files.

Click here to view code image

```
static List<String> readAllLines(Path path) throws IOException
static List<String> readAllLines(Path path, Charset cs) throws IOException
```

Read all *lines* from the file denoted by the specified `path`, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. These methods are not recommended for reading large files, as these can result in a lethal `java.lang.OutOfMemoryError`.

The code at (5) to (8) in Example 21.2 illustrates methods of the `Files` class that directly write and read character data to a file denoted by a `Path` object.

The `write()` method at (5) writes an `Iterable` (in this case, the `List` of `String`) to the file in one operation. It automatically terminates each string written with a newline.

Click here to view code image

```
Files.write(path, lines);                                        // (5)
```

The `writeString()` method at (6) writes the contents of a single `CharSequence` (in this case, the string `joinedLines`) to the file. The strings in the `lines` list are joined with an appropriate line separator by the `String.join()` method. The end result written to the file is again lines of text.

Click here to view code image

```
String joinedLines = String.join(System.lineSeparator(), lines);
Files.writeString(path, joinedLines);                             // (6)
```

The `readString()` method at (7) reads the *whole* file in one operation. It returns all characters read in a string, *including any line separators*.

Click here to view code image

```
String allContent = Files.readString(path);                      // (7)
```

The `readAllLines()` method at (8) reads all *text lines* in the file in one operation, returning the lines read in a `List` of `String`.

Click here to view code image

```
lines = Files.readAllLines(path);                                 // (8)
```

The methods in the `Files` class for writing and reading directly from a file denoted by a `Path` object should be used with care, as they might not scale up when handling large files. This is especially the case regarding the `readString()` and `readAll-Lines()` methods that read the whole file in one fell swoop. A better solution for reading text files using streams is provided later in the chapter (p. 1345).

Reading and Writing Bytes

The `Files` class also provides methods for reading and writing *bytes* to files. These methods that can be categorized as follows:

- Methods that create low-level *byte* I/O streams (`InputStream`, `OutputStream`) chained to a `Path` object that denotes a file. The methods of the I/O streams can then be used to read and write bytes to the file.
- Methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object.

Reading and Writing Bytes Using I/O Streams

The `newInputStream()` and `newOutputStream()` methods of the `Files` class create an input stream and an output stream, respectively, that are chained to a `Path` object denoting a file. Interoperability between I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing data to a file (§20.2, p. 1234).

Click here to view code image

```
static InputStream newInputStream(Path path, OpenOption... options)
                                 throws IOException
```

Opens a file denoted by the specified `path`, and returns an input stream to read from the file. No options implies the `READ` option.

Click here to view code image

```
static OutputStream newOutputStream(Path path, OpenOption... options)
                    throws IOException
```

Opens or creates a file denoted by the specified `path`, and returns an output stream that can be used to write bytes to the file. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Previously we have seen how the `copy()` methods of the `Files` class use byte I/O streams for reading and writing bytes to files (p. 1311).

The code at (1) in Example 21.3 is a reworking of Example 20.1, p. 1237, to copy bytes from a source file to a destination file using an explicit byte buffer. The main difference is that the input stream and the output stream on the respective files are created by the `newInputStream()` and `newOutputStream()` methods of the `Files` class, based on `Path` objects that denote the files, rather than on file I/O streams. As before, the methods `read()` and `write()` of the `InputStream` and `OutputStream` classes, respectively, are used to read and write the bytes from the source file to the destination file using a byte buffer.

Example 21.3 *Reading and Writing Bytes*

Click here to view code image

```
import java.io.*;
import java.nio.file.*;

public class ReadingWritingBytes {
  public static void main(String[] args) {
    // Source and destination files:
    Path srcPath  = Path.of("project", "source.dat");
    Path destPath = Path.of("project", "destination.dat");

    try (InputStream is = Files.newInputStream(srcPath);                    //
         OutputStream os = Files.newOutputStream(destPath))  {
      byte[] buffer = new byte[1024];
      int length = 0;

      while((length = is.read(buffer, 0, buffer.length)) != -1) {
        os.write(buffer, 0, length);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    try {
      // Reads the file contents into an array of bytes:
      byte[] allBytes = Files.readAllBytes(srcPath);                        //

      // Writes an array of bytes to a file:
      Files.write(destPath, allBytes);                                      //
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Reading and Writing Bytes Using Path Objects

The `Files` class provides methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object, without the need to specify a file I/O stream. The method `readAllBytes()` reads all bytes from a file into a `byte` array in one operation, and the method `write()` writes the bytes in a `byte` array to a file. These methods also close the file when done.

> Click here to view code image
>
> ```
> static byte[] readAllBytes(Path path) throws IOException
> ```
>
> Reads all the bytes from the file denoted by the specified `path`. The bytes are returned in a `byte` array.
>
> Click here to view code image
>
> ```
> static Path write(Path path, byte[] bytes, OpenOption... options)
>             throws IOException
> ```
>
> Writes bytes in a `byte` array to the file denoted by the specified `path`. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

The code at (2) and (3) in Example 21.3 shows yet another example of copying the contents of a source file to a destination file. The `readAllBytes()` and `write()` methods accomplish the task in a single call to each method.

Click here to view code image

```
byte[] allBytes = Files.readAllBytes(srcPath);                        // (2)
...
Files.write(destPath, allBytes);                                      // (3)
```

Note that these methods are meant for simple cases, and not for handling large files, as data is handled using an array of bytes.

21.6 Managing File Attributes

Useful metadata is associated with directory entries in a file system —for example, the file permissions that indicate whether the entry is readable or writable, or whether it is a symbolic link, and its size. Such metadata in the file system is often referred to as *file attributes*. Managing file attributes is a separate concern from the data that is stored in files.

There are basically two approaches provided by the NIO.2 API for managing file attributes:

- Accessing *individual file attributes* associated with a directory entry in the file system (p. 1321)
- Accessing a *set of file attributes* associated with a directory entry in the file system as a *bulk operation* (p. 1328)

Accessing Individual File Attributes

The `Files` class provides a myriad of static methods to access individual file attributes of a directory entry. It is a good idea to consult the code in Example 21.4 as we take a closer look at the relevant methods in this subsection. Since methods in the `Files` class can throw an `IOException`, the `main()` method specifies a `throws` clause with this exception.

Example 21.4 *Accessing Individual Attributes*

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

import static java.lang.System.out;
import static java.nio.file.attribute.PosixFilePermission.*;

public class IndividualFileAttributes {

  public static void main(String[] args) throws IOException {

    Path fPath = Path.of("project", "src", "pkg", "Main.java");
    out.println("File: " + fPath);

    out.println("Accessing Individual File Attributes:");
    out.println("size file (bytes): " + Files.size(fPath));                //
    out.println("isDirectory:       " + Files.isDirectory(fPath));         //
    out.println("isRegularFile:     " + Files.isRegularFile(fPath));       //
    out.println("isSymbolicLink:    " + Files.isSymbolicLink(fPath));      //
    out.println();

    out.println("isReadable:        " + Files.isReadable(fPath));          //
    out.println("isWritable:        " + Files.isWritable(fPath));          //

    out.println("isExecutable:      " + Files.isExecutable(fPath));        //
    out.println("isHidden:          " + Files.isHidden(fPath));            //
    out.println();

    out.println("getLastModifiedTime: " + Files.getLastModifiedTime(fPath));//
    out.println("getOwner:            " + Files.getOwner(fPath));          //
    out.println();

    // Get the POSIX file permissions for the directory entry:
    Set<PosixFilePermission> filePermissions
        = Files.getPosixFilePermissions(fPath);                            //
    out.println("getPosixFilePermissions (set): " + filePermissions);      //
    out.println("getPosixFilePermissions (string): "
              + PosixFilePermissions.toString(filePermissions));           //

    // Get the group of the directory entry:
    out.println("getAttribute-group:  " + Files.getAttribute(fPath,         //
                                                             "posix:group"));
    out.println();

    // Update last modified time for the directory entry.
    long currentTime = System.currentTimeMillis();
    FileTime timestamp = FileTime.fromMillis(currentTime);
    Files.setLastModifiedTime(fPath, timestamp);

    // Set new owner for the directory entry.
    FileSystem fs = fPath.getFileSystem();   // File system that created the p
    UserPrincipalLookupService upls
        = fs.getUserPrincipalLookupService();// Obtain service to look up user
    UserPrincipal user = upls.lookupPrincipalByName("khalid"); // User lookup.
    Files.setOwner(fPath, user);                               // Set user.

    // Set POSIX file permissions for the directory entry:
    Set<PosixFilePermission> newfilePermissions
        = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, GROUP_WRITE);   // (
    //Set<PosixFilePermission> newfilePermissions
    //  = PosixFilePermissions.fromString("rw-rw----");                   // (
    Files.setPosixFilePermissions(fPath, newfilePermissions);             // (
    filePermissions = Files.getPosixFilePermissions(fPath);
    out.println("getPosixFilePermissions (set): " + filePermissions);
    out.println("getPosixFilePermissions (string): "
          + PosixFilePermissions.toString(filePermissions));

    // Setting the value of a file attribute by its attribute name.
    Files.setAttribute(fPath, "lastAccessTime", timestamp);               // (
  }
}
```

Possible output from the program:

Click here to view code image

```
File: project/src/pkg/Main.java
Accessing Individual File Attributes:
size file (bytes): 13

isDirectory:       false
isRegularFile:     true
isSymbolicLink:    false

isReadable:        true
isWritable:        true
isExecutable:      false
isHidden:          false

getLastModifiedTime: 2021-08-06T10:28:47.416033Z
getOwner:            khalid

getPosixFilePermissions (set): [OTHERS_READ, OWNER_WRITE, OWNER_READ, GROUP_RE
getPosixFilePermissions (string): rw-r--r--
getAttribute-group:  admin
```

```
getPosixFilePermissions (set): [GROUP_WRITE, OWNER_WRITE, OWNER_READ, GROUP_RE
getPosixFilePermissions (string): rw-rw----
```

Determining the File Size

The method `size()` in the `Files` class is called at (1) in Example 21.4 to determine the size of the file denoted by the `Path` object. If the `Path` object denotes a directory, the method returns the size of the directory file and *not* the size of entries in the directory.

> Click here to view code image
>
> ```
> static long size(Path path) throws IOException
> ```
>
> Returns the size of a file (in bytes). The size of files that are not regular files is unspecified, as it is implementation specific.

Determining the Kind of Directory Entry

The following methods in the `Files` class are called at (2), (3), and (4) in Example 21.4 to determine what kind of directory entry is denoted by the `Path` object.

> Click here to view code image
>
> ```
> static boolean isDirectory(Path path, LinkOption... options)
> static boolean isRegularFile(Path path, LinkOption... options)
> static boolean isSymbolicLink(Path path)
> ```
>
> Return `true` if the directory entry is a directory, a regular file, or a symbolic link, respectively. They return `false` if the directory entry does not exist, or is not of the expected kind, or it is not possible to determine what kind of directory entry it is. In the first two methods, symbolic links are followed by default, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Determining File Accessibility

The methods in the `Files` class shown below are called at (5), (6), (7), and (8) in Example 21.4, respectively, to determine accessibility of the directory entry denoted by the `Path` object.

> Click here to view code image
>
> ```
> static boolean isReadable(Path path)
> static boolean isWritable(Path path)
> static boolean isExecutable(Path path)
> ```

Test whether a file is readable, writable, or executable, respectively. The file must exist and the JVM must have the appropriate privileges to access the file.

Note that the result returned by these method is immediately outdated. The outcome of subsequent attempts to access the file is not guaranteed, as concurrently running threads might change the conditions after the method returns.

Click here to view code image

```
static boolean isHidden(Path path) throws IOException
```

Determines whether or not a file is considered *hidden*. The exact definition of a hidden file is platform specific. On Unix platforms, files whose name begins with a period character (`.`) are considered to be hidden.

Timestamp for Last Modification Time

Three different timestamps are associated with a directory entry, whose purpose is evident from their names: *last modified time*, *last access time*, and *creation time*. The timestamps are represented by the `java.nio.file.attribute.FileTime` class that provides the following methods for interoperability with `Instant` objects and with `long` values in milliseconds.

Click here to view code image

```
class java.nio.file.attribute.FileTime

static FileTime from(Instant instant)
static FileTime fromMillis(long value)
```

These static methods create a `FileTime` object representing the same point of time value on the timeline as the specified `Instant` object, or a `FileTime` object from the `long value` that specifies the number of milliseconds since the epoch (`1970-01-01T00:00:00Z`), respectively.

```
Instant toInstant()
long toMillis()
```

These instance methods convert this `FileTime` object to an `Instant` or to a `long` value in milliseconds from the epoch, respectively.

The `Files` class only provides static methods to read and update the last modified time of a directory entry. In Example 21.4, the statement at (9) prints the last modified time of the directory entry. The code at (15) sets the last modified time of the directory entry to the current time.

> Click here to view code image
>
> ```
> static FileTime getLastModifiedTime(Path path, LinkOption... options)
>                                     throws IOException
> static Path     setLastModifiedTime(Path path, FileTime time)
>                                     throws IOException
> ```
>
> Return or update the timestamp for the last modified time attribute of a directory entry, respectively. The timestamp is represented by the class `java.nio .file.attribute.FileTime`.
>
> The first method follows symbolic links by default, unless the constant `Link-Option.NOFOLLOW_LINKS` is specified.

Accessing the Owner

The `Files` class only provides static methods to get and set the *owner* of a directory entry (i.e., one with a user account and appropriate access permissions). In Example 21.4, the statement at (10) prints the name of the owner of the directory entry. The code at (16) executes the necessary steps to obtain a user that can be set as the owner of the directory entry. This involves querying the file system to obtain the user look service and using the service to look up the user by name. We leave it to the reader to discover the exciting details from the API of the classes involved.

> Click here to view code image
>
> ```
> static UserPrincipal getOwner(Path path, LinkOption... options)
> static Path setOwner(Path path, UserPrincipal owner)
> ```
>
> Return or update the owner of a file, respectively.

Handling File Permissions

For a directory entry, POSIX-based file systems (*Portable Operating System Interface*) typically define *read*, *write*, and *execute* permissions for the *owner*, the *group* that the owner belongs to, and for *others*. In Java, these nine permissions are represented by the enum type `PosixFilePermission` (Table 21.10).

A human-readable form of file permissions affords interoperability with the enum type `PosixFilePermission`. This form is specified as a string of nine characters, where characters are interpreted as three permission groups of three characters. From the start of the string, the first permission group, the second permission group, and the third permission group specify the permissions for the owner, the group, and others, respectively. Each permission group is defined by the following pattern:

```
(r|-)(w|-)(x|-)
```

that is comprised of three groupings, where each grouping `(a|b)` is interpreted as either `a` or `b`. For example, `rwx` and `---` are valid permissions groups, but `w_w` and `xwr` are not. The characters `r`, `w`, and `x` stand for read, write, and execute permissions, respectively, and the character `-` indicates that the permission corresponding to the position of the character `-` is not set.

The set of file permissions created by the following statement:

Click here to view code image

```
Set<PosixFilePermission> permSet1
      = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);
```

is equivalent to the permissions in the string `"rw-r--r--"`.

The utility class `PosixFilePermissions` provides methods for converting between the two forms of specifying file permissions.

Table 21.10 *POSIX File Permissions*

Enum type java.nio.file.attrib	**Description**
`OWNER_EXECUTE`	Execute/search permission, owner
`OWNER_READ`	Read permission, owner
`OWNER_WRITE`	Write permission, owner
`GROUP_EXECUTE`	Execute/search permission, group
`GROUP_READ`	Read permission, group
`GROUP_WRITE`	Write permission, group
`OTHERS_EXECUTE`	Execute/search permission, others
`OTHERS_READ`	Read permission, others
`OTHERS_WRITE`	Write permission, others

Following are methods from the utility class `java.nio.file.attribute.PosixFilePermissions`:

Click here to view code image

```
static Set<PosixFilePermission> fromString(String permStr)
```

Returns the set of permissions corresponding to a given `String` representation. The `permStr` parameter is a `String` representing the permissions, as explained earlier.

Click here to view code image

```
static String toString(Set<PosixFilePermission> perms)
```

Returns the `String` representation of a set of permissions.

Click here to view code image

```
  static FileAttribute<Set<PosixFilePermission>>
         asFileAttribute(Set<PosixFilePermission> perms)
```

Creates a `FileAttribute`, encapsulating a copy of the given file permissions, suitable for passing to methods that create files and directories (p. 1339).

The `getPosixFilePermissions()` and `setPosixFilePermissions()` methods of the `Files` class can be used to retrieve and update file permissions of a directory entry, as shown at (11) and (19), respectively. The methods `toString()` and `fromString()` of the `PosixFilePermissions` class at (13) and (18b) convert between a set of `PosixFilePermission` and a string representation of file permissions, respectively. Note that (18a) and (18b) define the same set of file permissions.

Click here to view code image

```
// Get the POSIX file permissions for the directory entry:
Set<PosixFilePermission> filePermissions
    = Files.getPosixFilePermissions(fPath);                            // (11
out.println("getPosixFilePermissions (set):    " + filePermissions);    // (12
out.println("getPosixFilePermissions (string): "
           + PosixFilePermissions.toString(filePermissions));          // (13
...
// Set POSIX file permissions for the directory entry:                    (17
Set<PosixFilePermission> newfilePermissions
         = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);// (18
//Set<PosixFilePermission> newfilePermissions
//       = PosixFilePermissions.fromString("rw-r--r--");               // (18
Files.setPosixFilePermissions(fPath, newfilePermissions);              // (19
```

The following methods from the utility class `java.nio.file.Files` can be used for retrieving and updating the POSIX-specific file permissions of a directory entry:

Click here to view code image

```
static Set<PosixFilePermission>
       getPosixFilePermissions(Path path, LinkOption... options)
                                throws IOException
```

Returns POSIX permissions of a directory entry as a set of

> enum type `PosixFilePermission`. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.
>
> Click here to view code image
>
> ```
> static Path setPosixFilePermissions(Path path,
>                                     Set<PosixFilePermission< perms)
>                                     throws IOException
> ```
>
> Sets the POSIX permissions of a directory entry, given by the parameter `perms`.

Accessing File Attributes through View and Attribute Names

The `getAttribute()` and `setAttribute()` methods of the `Files` class are general methods that can be used to read and update any file attribute by its name. These methods are useful when the `Files` class does not provide a specialized method for a particular file attribute.

The statement at (14) in Example 21.4 prints the group of the directory entry. The full attribute name of the group attribute is specified as `"posix:group"`, as the group attribute can be accessed via the POSIX file attribute view (p. 1336).

The statement at (20) in Example 21.4 sets the last access time of the directory entry to the current time. The attribute named `"lastAccessTime"` can be accessed via the basic file attribute view that is implied by default (p. 1334). The value of this file attribute is the `FileTime` object denoted by the `timestamp` reference.

> Click here to view code image
>
> ```
> static Object getAttribute(Path path, String attribute,
>                            LinkOption... options) throws IOException
> static Path   setAttribute(Path path, String attribute, Object value,
>                            LinkOption... options) throws IOException
> ```
>
> Read and set, respectively, the value of a file `attribute` of a directory entry denoted by the given `Path` object. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.
>
> The `attribute` parameter has the general format:
>
> ```
> view-name:attribute-name
> ```
>
> where the *view-name* can be omitted. Typical view names are

> `"basic"`, `"posix"`, and `"dos"`. Omitting the *view-name* defaults to `"basic"`. The *attribute-name* is the name of the file attribute —for example, `"lastModifiedTime"` or `"lastAccess-Time"`. Attribute views are discussed in detail later (p. 1328).
>
> The second method sets the file `attribute` to the `value` parameter.

Bulk Operations to Retrieve File Attributes

Accessing file attributes individually raises two concerns:

- Accessing an individual file attribute incurs a cost, and frequent such accesses can adversely impact performance.
- File attributes are very much file-system-specific, and therefore not conducive to generalizing over different file systems.

To address these concerns, the NIO.2 API provides *bulk operations* to retrieve file attributes—thus avoiding retrieval of individual file attributes and only handling file-system-specific attributes.

The following three approaches can be used to access file attributes of a directory entry in a file system:

- The methods of the `Files` class can be used to access individual file attributes of a directory entry denoted by a `Path`, as discussed earlier (p. 1321).
- A *bulk operation* using the `readAttributes()` method of the `Files` class can be used to retrieve a *set of file attributes* into a *read-only object* that implements *a file attributes interface*. This read-only object acts as a repository for file attribute values pertaining to the directory entry whose `Path` object was specified in the call to the `readAttributes()` method. The interface `BasicFileAttributes`, and its subinterfaces `PosixFileAttributes` and `DosFileAttributes`, define methods to read the retrieved file attribute values (Table 21.11, p. 1330).

  In the code below at (1), the set of file attributes to read for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributes.class`.

  Click here to view code image

```
BasicFileAttributes bfa = Files.readAttributes(path,                  /
                                               BasicFileAttributes.clas
```

The retrieved values of the file attributes are accessible by querying the `BasicFileAttributes` object that is returned by the `readAttributes()` method (p. 1330).

- A *bulk operation* using the `getFileAttributeView()` method of the `Files` class can be used to retrieve *a set of file attributes* into an *updatable file attribute view* that acts as a repository object. This object can be used to read or update selected file attributes pertaining to the directory entry whose `Path` object was specified in the call to the `getFileAttributeView()` method. The interface `BasicFileAttributeView`, and its subinterfaces `PosixFileAttributeView` and `DosFileAttributeView`, define methods to read and update the retrieved file attributes (Table 21.12, p. 1334).

  In the code below at (6), the set of file attributes to retrieve for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributeView.class`.

  Click here to view code image

  ```
  BasicFileAttributeView bfaView = Files.getFileAttributeView(path,
                                                    BasicFileAttributeVie
  ```

  The retrieved file attributes can be read and updated by querying the `Basic-FileAttributeView` object that is returned by the `getFileAttributeView()` method (p. 1334).

Details of the API for the `readAttributes()` and the `getFileAttributeView()` methods in the `Files` class are given below. By default, these methods follow symbolic links, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

> Click here to view code image
>
> ```
> static <A extends BasicFileAttributes> A
>        readAttributes(Path path, Class<A> type, LinkOption... options)
>        throws IOException
> ```
>
> Reads a *set of read-only file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. The file attributes to retrieve are determined by the type parameter `A`. The parameter `type` is the `Class<A>` of the file attributes required to retrieve. The type parameter `A` is typically the

interface `BasicFileAttributes`, or one of its subinterfaces `PosixFileAttributes` or `DosFileAttributes` (Table 21.11, p. 1330).

Click here to view code image

```
static <V extends FileAttributeView> V
       getFileAttributeView(Path path, Class<V> type, LinkOption... options)
```

Reads a *set of file attributes* of a file as a *bulk operation*. It returns a *file attribute view* of a given type, which can be used to *read or update* the retrieved file attribute values. The file attributes to retrieve are determined by the type parameter `V`. The parameter `type` is the `Class<V>` of the file attributes required to retrieve. The type parameter `V` is typically the interface `BasicFileAttributeView`, or one of its subinterfaces `PosixFileAttributeView` or `DosFileAttributeView`— that are all subinterfaces of the `FileAttributeView` interface (Table 21.12, p. 1334).

Click here to view code image

```
static Map<String,Object> readAttributes(Path path, String attributes,
                                          LinkOption... options)
                                          throws IOException
```

Reads a *set of file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. It returns a map of file attribute names and their values.

The `attributes` parameter of type `String` has the general format:

```
view-name:attribute-list
```

where the *view-name* can be omitted, and *attribute-list* is a comma-separated list of attribute names. Omitting the *view-name* defaults to `"basic"`. For example, `"*"` will read all `BasicFileAttributes`, and `"lastModifiedTime,lastAccessTime"` will read only last modified time and last access time attributes.

Table 21.11 summarizes the interfaces that provide read-only access to file attribute values. The `BasicFileAttributes` interface defines the basic set of file attributes that are common to many file systems. Its two subinterfaces `PosixFileAttributes` and `DosFileAttributes` define *additional* file attributes associated with POSIX-based and DOS-based file systems, respectively. An object of the appropriate file attributes interface pertaining to a specific directory entry is returned by the `Files.read-Attributes()` method.

Table 21.11 *Interfaces for Read-Only Access to File Attributes*

Read-only file attribute interfaces in `java.nio.file.attribute` package	**Note that when an object of the read-only file attributes interface is created, it is opened on a specific directory entry in the file system. The object provides information about file attributes associated with this directory entry. The methods in these interfaces do not throw a checked exception.**
`BasicFileAttributes`	Provides *read-only* access to a basic set of file attributes that are common to many file systems (p. 1330).
`PosixFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with POSIX-based file systems (p. 1332).
`DosFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with DOS/ Windows-based file systems (p. 1333).

The BasicFileAttributes Interface

The methods of the `java.nio.file.attribute.BasicFileAttributes` interface reflect the basic set of file attributes that are common to most file systems.

> Click here to view code image
>
> ```
> interface java.nio.file.attribute.BasicFileAttributes
> ```
>
> ```
> long size()
> ```
>
> Returns the size of the directory entry (in bytes).
>
> ```
> boolean isDirectory()
> boolean isRegularFile()
> boolean isSymbolicLink()
> boolean isOther()
> ```
>
> Determine whether the directory entry is of a specific kind.
>
> Click here to view code image
>
> ```
> FileTime lastModifiedTime()
> FileTime lastAccessTime()
> FileTime creationTime()
> ```
>
> Return the appropriate timestamp for the directory entry.

The method `printBasicFileAttributes()` of the utility class `FileUtils`, shown below, prints the values of the basic file attributes by calling the relevant methods on the

`BasicFileAttributes` object that is passed as a parameter.

Click here to view code image

```
// Declared in utility class FileUtils.
public static void printBasicFileAttributes(BasicFileAttributes bfa) {
  out.println("Printing basic file attributes:");
  out.println("lastModifiedTime: " + bfa.lastModifiedTime());
  out.println("lastAccessTime:   " + bfa.lastAccessTime());
  out.println("creationTime:     " + bfa.creationTime());

  out.println("size:             " + bfa.size());
  out.println("isDirectory:      " + bfa.isDirectory());
  out.println("isRegularFile:    " + bfa.isRegularFile());
  out.println("isSymbolicLink:   " + bfa.isSymbolicLink());
  out.println("isOther:          " + bfa.isOther());
  out.println();
}
```

The code below obtains a `BasicFileAttributes` object at (1) that pertains to the file denoted by the `path` reference. The `printBasicFileAttributes()` method is called at (2) with this `BasicFileAttributes` object as the parameter.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributes bfa = Files.readAttributes(path,                  // (1)
                                               BasicFileAttributes.class);
FileUtils.printBasicFileAttributes(bfa);                              // (2)
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Printing basic file attributes:
lastModifiedTime: 2021-07-23T10:15:34.854Z
lastAccessTime:   2021-07-23T10:16:33.166281Z
creationTime:     2021-07-20T23:03:58Z
size:             116
isDirectory:      false

isRegularFile:    true
isSymbolicLink:   false
isOther:          false
```

Note that the basic file attributes in the `BasicFileAttributes` object are read-only, as the `BasicFileAttributes` interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the `Files` class (p. 1321).

The PosixFileAttributes Interface

As the `PosixFileAttributes` interface is a subinterface of the `BasicFileAttributes` interface, a `PosixFileAttributes` object has both the basic set of file attributes and the POSIX-specific file attributes.

> Click here to view code image
>
> ```
> interface java.nio.file.attribute.PosixFileAttributes
>           extends BasicFileAttributes
> ```
>
> ```
>   UserPrincipal owner()
>   GroupPrincipal group()
> ```
>
> Return the owner or the group of the directory entry, represented by the interface `java.nio.file.attribute.UserPrincipal` and its subinterface `java.nio.file .attribute.GroupPrincipal`, respectively.
>
> Click here to view code image
>
> ```
> Set<PosixFilePermission> permissions()
> ```
>
> Returns a set with a copy of the POSIX permissions for the directory entry. Permissions are defined by the enum type `PosixFilePermission` discussed earlier in this chapter (Table 21.10, p. 1326).

The methods of the subinterface `PosixFileAttributes` augment the basic set of file attributes with the following POSIX-specific attributes: owner, group, and file permissions. The method `printPosixFileAttributes()` in the utility class `FileUtils`, shown below, prints the values of the POSIX-specific file attributes by calling the relevant methods on the `PosixFileAttributes` object that is passed as a parameter.

Click here to view code image

```
// Declared in the utility class FileUtils.
public static void printPosixFileAttributes(PosixFileAttributes pfa) {
  out.println("Printing POSIX-specific file attributes:");
  UserPrincipal user = pfa.owner();
  GroupPrincipal group = pfa.group();
  Set<PosixFilePermission> permissions = pfa.permissions();
  String perms = PosixFilePermissions.toString(permissions);

  out.println("owner:            " + user);
  out.println("group:            " + group);
  out.println("permissions:      " + perms);
  out.println();
}
```

The code below obtains a `PosixFileAttributes` object at (3) that pertains to the file denoted by the `path` reference. Both the `printBasicFileAttributes()` method and the `printPosixFileAttributes()` method are called at (4) and (5), respectively, with this `PosixFileAttributes` object as the parameter. The call to the `printBasicFile-Attributes()` method at (4) will print the basic file attributes in the `PosixFile-Attributes` object.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributes pfa = Files.readAttributes(path,                          // (3)
                                               PosixFileAttributes.class);
FileUtils.printBasicFileAttributes(pfa);                                      // (4)
FileUtils.printPosixFileAttributes(pfa);                                      // (5)
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Printing basic file attributes:
...
Printing POSIX-specific file attributes:
owner:            javadude
group:            admin
permissions:      rw-r--r--
```

Note that both the basic and the POSIX-specific file attributes in the `PosixFileAttributes` object are readonly, as the `PosixFileAttributes` interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the `Files` class (p. 1321).

The `DosFileAttributes` *Interface*

As the `DosFileAttributes` interface is a subinterface of the `BasicFileAttributes` interface, a `DosFileAttributes` object has both the basic set of file attributes and the DOS-specific file attributes. Its usage is analogous to the `PosixFileAttributes` interface discussed earlier (p. 1332).

Click here to view code image

```
interface java.nio.file.attribute.DosFileAttributes
          extends BasicFileAttributes

boolean isReadOnly()
boolean isSystem()
boolean isArchive()
boolean isHidden()
```

Determine whether the file entry is of a specific kind.

File Attribute Views

Table 21.12 summarizes the *file attribute views* that different interfaces provide for *readable or updatable* access to file attributes, in contrast to the file attributes interfaces in Table 21.11, p. 1330, that allow readonly access. The `BasicFileAttribute-View` interface allows access to the basic set of file attributes that are common to many file systems. Its two subinterfaces, `PosixFileAttributeView` and `DosFileAttributeView`, additionally allow access to file attributes associated with POSIX-based and DOS-based file systems, respectively.

An object of the appropriate view interface pertaining to a specific directory entry is returned by the `Files.getFileAttributeView()` method. All file interface views provide a `readAttributes()` method that returns the readonly file attributes object associated with the view.

Table 21.12 *Selected File Attribute Views*

Updatable file attribute view `java.nio.file.attribute` **package**	**Note that when the view is created, it is opened on a specific directory entry in the file system. The view provides information about file attributes associated with this directory entry.**
`AttributeView`	Can read or update non-opaque values associated with directory entries in a file system.
`FileAttributeView extends AttributeView`	Can read or update file attributes.
`FileOwnerAttributeView extends FileAttributeView`	Can read or update the owner.
`BasicFileAttributeView extends FileAttributeView` *Corresponding readonly file attributes interface:* `BasicFileAttributes`	Can read or update a basic set of file attributes. Can obtain a readonly `BasicFileAttributes` object via the view. Can set a timestamp for when the directory entry was last modified, last accessed, and created. (p. 1334)
`PosixFileAttributeView extends BasicFileAttributeView, FileOwnerAttributeView` *Corresponding read-only file attributes interface:* `PosixFileAttributes`	Can read or update POSIX file attributes. Can obtain a read-only `PosixFileAttributes` object via the view. Can set group and file permissions, and update the owner. (p. 1336)
`DosFileAttributeView extends BasicFileAttributeView` *Corresponding read-only file attributes interface:* `DosFileAttributes`	Can read or update DOS file attributes. Can obtain a read-only `DosFileAttributes` object via the view. Can set archive, hidden, read-only, and system attributes. (p. 1338)

The `BasicFileAttributeView` Interface

The `java.nio.file.attribute.BasicFileAttributeView` interface defines a file attribute view for the basic set of file attributes.

> Click here to view code image
>
> ```
> interface BasicFileAttributeView extends FileAttributeView
> ```
>
> ```
> String name()
> ```
>
> Returns the name of the attribute view, which in this case is the string `"basic"`.
>
> Click here to view code image
>
> ```
> BasicFileAttributes readAttributes()
> ```
>
> Reads the basic file attributes as a bulk operation. The `BasicFileAttributes` object can be used to read the values of the basic file attributes (p. 1330). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).
>
> Click here to view code image
>
> ```
> void setTimes(FileTime lastModifiedTime,
>               FileTime lastAccessTime,
>               FileTime createTime)
> ```
>
> Updates any or all timestamps for the file's last modified time, last access time, and creation time attributes. If any parameter has the value `null`, the corresponding timestamp is not changed. Note that apart from the `Files.setLast-Modified()` method, there are no methods in the `Files` class for the last access and creation times for a directory entry.

The code below obtains a `BasicFileAttributeView` object at (6) that pertains to the file denoted by the `path` reference. A `BasicFileAttributes` object is obtained at (7), providing read-only access to the basic file attributes, whose values are printed by calling the `printBasicFileAttributes()` method. The last modified time of the directory entry is explicitly read by calling the `lastModifiedTime()` method of the `BasicFileAttributes` object.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                     BasicFileAttributeView.class);
System.out.printf("Using view: %s%n", bfaView.name());

// Reading the basic set of file attributes:                           (7)
BasicFileAttributes bfa2 = bfaView.readAttributes();
FileUtils.printBasicFileAttributes(bfa2);
```

```
  FileTime currentLastModifiedTime = bfa2.lastModifiedTime();

  // Updating timestamp for last modified time using view:                  (8)
  long newLMTinMillis = currentLastModifiedTime.toMillis() + 15*60*1000L;
  FileTime newLastModifiedTime = FileTime.fromMillis(newLMTinMillis);
  bfaView.setTimes(newLastModifiedTime, null, null);

  // Reading the updated last modified time:                                (9)
  out.println("updated lastModifiedTime (incorrect): "
                                          + bfa2.lastModifiedTime());   // (10)
  out.println("updated lastModifiedTime: "
                                    + Files.getLastModifiedTime(path)); // (11)
  out.println("updated lastModifiedTime: " + Files.getAttribute(path,    // (12)
                                                 "basic:lastModifiedTime"));
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Using view: basic
Printing basic file attributes:
...
lastModifiedTime: 2021-07-26T15:15:46.813Z
...
updated lastModifiedTime (incorrect): 2021-07-26T15:15:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
```

The `BasicFileAttributeView` object allows the last modified, last access, and creation times of the directory entry to be updated by calling the `setTimes()` method. The code at (9) shows how the last modified time of the directory entry can be updated to a new value via the view. A `FileTime` object is created representing the new last modified time by first converting the current last modified time to milliseconds and incrementing it by 15 minutes. In the call to the `setTimes()` method, only the last modified time is specified. The other timestamps are specified as `null`, indicating that they should not be changed.

In order to verify the new last modified time, we might be tempted to use the current `BasicFileAttributes` object associated with the view, but its copies of the file attribute values are not updatable. We can create a new `BasicFileAttributes` object that reflects the new values of the file attributes, or alternately use the `getLastModifiedTime()` or the `getAttribute()` methods of the `Files` class, as shown at (11) and (12), respectively.

The PosixFileAttributeView Interface

As the `PosixFileAttributeView` interface is a subinterface of the `BasicFileAttribute-View` interface, it allows both the basic set of file attributes and the POSIX-specific file attributes to be read and updated.

> Click here to view code image
>
> ```
>   interface PosixFileAttributeView
>             extends BasicFileAttributeView, FileOwnerAttributeView
>
>   String name()
> ```
>
> Returns the name of the attribute view, which in this case is the string `"posix"`.
>
> Click here to view code image
>
> ```
> PosixFileAttributes readAttributes()
> ```
>
> Retrieves the basic and POSIX-specific file attributes as a bulk operation into a `PosixFileAttributes` object whose methods can be used to read the values of these file attributes (p. 1332). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).
>
> Click here to view code image
>
> ```
> void setGroup(GroupPrincipal group) throws IOException
> ```
>
> Updates the group of the directory entry. Note that there is no analogous method in the `Files` class for handling the group.
>
> Click here to view code image
>
> ```
> void setPermissions(Set<PosixFilePermission> perms)
> ```
>
> Updates the file permissions. This method is analogous to the `Files.setPosixFilePermissions()` method (p. 1325).

The `PosixFileAttributeView` interface extends the `java.nio.file.attribute.File-OwnerAttributeView` interface that defines the methods for reading and updating the owner of the directory entry. See also analogous methods relating to ownership in the `Files` class (p. 1325).

> Click here to view code image
>
> ```
> interface java.nio.file.attribute.FileOwnerAttributeView
>           extends FileAttributeView
>
> UserPrincipal getOwner() throws IOException
> void          setOwner(UserPrincipal owner) throws IOException
> ```
>
> Return or update the owner of a directory entry, respectively. These methods are analogous to the methods in the `Files`

class (p. 1325).

A `PosixFileAttributeView` object can thus read both the basic set of file attributes and the POSIX-specific file attributes, and can update the owner, group, file permissions, and timestamps for the last modified, last access, and creation times for a directory entry.

The code below obtains a `PosixFileAttributeView` object at (13) that pertains to the file denoted by the `path` reference. The associated `PosixFileAttributes` object is obtained at (14), providing read-only access to the basic file attributes and the POSIX-specific file attributes, whose values are printed by calling the methods `printBasicFileAttributes()` and `printPosixFileAttributes()` in the utility class `FileUtils`, respectively.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributeView pfaView = Files.getFileAttributeView(path,      // (13)
                                              PosixFileAttributeView.class);
System.out.printf("Using view: %s%n", pfaView.name());

// Reading the basic + POSIX set of file attributes:                   // (14)
PosixFileAttributes pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);

// Updating owner and group file attributes using view.                // (15)
FileSystem fs = path.getFileSystem();
UserPrincipalLookupService upls = fs.getUserPrincipalLookupService();
UserPrincipal newUser = upls.lookupPrincipalByName("javadude");
GroupPrincipal newGroup = upls.lookupPrincipalByGroupName("admin");
pfaView.setOwner(newUser);
pfaView.setGroup(newGroup);

//Updating file permissions using view.                                // (16)
Set<PosixFilePermission> newPerms = PosixFilePermissions.fromString("r--r--r--
pfaView.setPermissions(newPerms);

//Updating last access time using view.                                // (17)
FileTime currentAccessTime = pfa2.lastAccessTime();
long newLATinMillis = currentAccessTime.toMillis() + 10*60*1000L;
FileTime newLastAccessTime = FileTime.fromMillis(newLATinMillis);
pfaView.setTimes(null, newLastAccessTime, null);

// Reading the updated file attributes:                                // (18)
pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);
```

The code from (15) to (17) shows how the `PosixFileAttributeView` object can be used to update various file attributes. Keep in mind that this view inherits from the `BasicFileAttributeView` and the

`FileOwnerAttributeView` interfaces.

The code at (15) updates the owner and the group of the directory entry via the view. An owner and a group are looked up in the appropriate lookup services, and updated by the `setOwner()` and `setGroup()` methods of the `PosixFileAttributeView` interface. See also corresponding methods in the `Files` class (p. 1325).

The code at (16) updates the file permissions of the directory entry via the view, analogous to the `Files.setPosixFilePermissions()` method (p. 1325). File permissions are set to read-only for the owner, the group, and other users.

The code at (17) updates only the last access time of the directory entry via the view, analogous to updating the last modified time via the `BasicFileAttributeView` object (p. 1334).

Updated file attribute values can be read using the appropriate methods of the `Files` class, or by obtaining a new `PosixFileAttributes` object, as shown at (18).

The DosFileAttributeView Interface

As the `DosFileAttributeView` interface is a subinterface of the `BasicFileAttributeView` interface, it allows both the basic set of file attributes and the DOS-specific file attributes to be read and updated. Its usage is analogous to the `PosixFileAttributeView` interface discussed earlier (p. 1336).

> Click here to view code image
>
> ```
> interface DosFileAttributeView extends BasicFileAttributeView
> ```
>
> ```
> String name()
> ```
>
> Returns the name of the attribute view, which in this case is the string `"dos"`.
>
> Click here to view code image
>
> ```
> DosFileAttributes readAttributes()
> ```
>
> Reads the basic file attributes as a bulk operation. The `DosFileAttributes` object can be used to read the values of the basic and DOS-specific file attributes (p. 1333). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328)
>
> Click here to view code image

```
void setReadOnly(boolean value)
void setSystem(boolean value)
void setArchive(boolean value)
void setHidden(boolean value)
```

> Update the value of the appropriate attribute. These methods are all implementation specific.

21.7 Creating Directory Entries

The `Files` class provides methods for creating files and directories. These methods can accept a variable arity parameter of type `FileAttribute<?>`. The interface `java.nio.file.attribute.FileAttribute<T>` defines an object that encapsulates the value of a file attribute that can be set when a file or a directory is created by these methods. For a POSIX-based file system, the `PosixFilePermissions.asFileAttribute()` method creates a `FileAttribute` that encapsulates a set of type `PosixFilePermission` (p. 1325).

The methods for creating regular files and directories throw a `FileAlreadyExists-Exception` if the directory entry with that name already exists. All methods for creating directory entries can throw an `IOException`, and should be called in a `try` block or the exception should be specified in a `throws` clause.

The steps to check whether the directory entry exists and to create the new directory entry if it does not exist are performed as a single *atomic operation.*

The following code calls the `printDirEntryInfo()` method in the utility class `FileUtils` on a path to print what kind of directory entry it denotes and its file permissions.

Click here to view code image

```
public static void printDirEntryInfo(Path path) throws IOException {
  String fmt = Files.isSymbolicLink(path)? "Symbolic link: %s%n":
      Files.isRegularFile(path)? "File: %s%n":
        Files.isDirectory(path)? "Directory: %s%n":
          "Directory entry: %s%n";
  out.printf(fmt, path);
  Set<PosixFilePermission> perms = Files.getPosixFilePermissions(path);
  String permStr = PosixFilePermissions.toString(perms);
  out.println(permStr);
}
```

Creating Regular and Temporary Files

The following methods of the `Files` class can be used to create regular and temporary files, and symbolic links.

> Click here to view code image
>
> ```
> static Path createFile(Path path, FileAttribute<?>... attrs)
>                        throws IOException
> ```
>
> Creates an empty file that is denoted by the `path` parameter, but fails by throwing a `FileAlreadyExistsException` if the file already exists. It does *not* create nonexistent parent directories, and throws a `NoSuchFileException` if any parent directory does not exist.
>
> Click here to view code image
>
> ```
> static Path createTempFile(String prefix, String suffix,
>                            FileAttribute<?>... attrs) throws IOException
> static Path createTempFile(Path dir, String prefix, String suffix,
>                            FileAttribute<?>... attrs) throws IOException
> ```
>
> Create an empty file in the *default temporary-file directory* or in the *specified directory* denoted by the `Path` object, respectively, using the specified `prefix` and `suffix` to generate its name. The default temporary-file directory and naming of temporary files is file-system-specific.
>
> Click here to view code image
>
> ```
> static Path createSymbolicLink(Path symbLink, Path target,
>                                FileAttribute<?>... attrs) throws IOException
> ```
>
> Creates a symbolic link to a target (*optional operation*). When the target is a *relative path* then file system operations on the target are relative to the path of the symbolic link. The target of the symbolic link need not exist. Throws a `FileAlreadyExistsException` if a directory entry with the same name as the symbolic link already exists.
>
> Click here to view code image
>
> ```
> static Path readSymbolicLink(Path symbLink) throws IOException
> ```
>
> Returns a `Path` object that denoted the target of a symbolic link (*optional operation*). The target of the symbolic link need not exist.

For creating files, we will use the following `FileAttribute` object denoted by the `fileAttr` reference (p. 1325). It specifies read and write permissions for the owner, but only read access for the group and others.

Click here to view code image

```
Set<PosixFilePermission> filePerms = PosixFilePermissions.fromString("rw-r--r-
FileAttribute<Set<PosixFilePermission>> fileAttr =
        PosixFilePermissions.asFileAttribute(perms);
```

Creating Regular Files

The `createFile()` method of the `Files` class fails to create a regular file if the file already exists, or the parent directories on the path do not exist. The code below at (1) creates a file with the path `project/docs/readme.txt` relative to the current directory under the assumption that the file name does not exist and the parent directories exist on the path. However, running this code repeatedly will result in a `FileAlreadyExistsException`, unless the file is deleted (e.g., calling `Files.deleteIfExists(regularFile)`) before rerunning the code.

Click here to view code image

```
try {
  Path regularFile  = Path.of("project", "docs", "readme.txt");
  Path createdFile1 = Files.createFile(regularFile, fileAttr);        // (1)
  FileUtils.printDirEntryInfo(createdFile1);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

```
File: project/docs/readme.txt
rw-r--r--
```

Creating Temporary Files

Programs usually create temporary files during execution, and delete such files when done in the interest of good housekeeping. The JVM defines a system property for the default temporary-file directory where temporary files are created, if a specific location is not specified. This location is printed by the code below at (1).

How the name of the temporary file is generated is file-system specific. The code at (2) creates a temporary file under the default temporary-file directory, that has default file permissions.

The code at (3) creates temporary files under a specific directory. How the specification of file name prefix and suffix affect the generated file name can be seen in the output, where the `null` value as the prefix omits the prefix, and the `null` value as the suffix appends the default file name extension `".tmp"`.

The NIO.2 API does not define a method to request that a file be deleted when the JVM terminates.

Click here to view code image

```
try {
  // System property that defines the default temporary-file directory.      (
  String tmpdir = System.getProperty("java.io.tmpdir");
  System.out.println("Default temporary directory: " + tmpdir);

  // Create under the default temporary-file directory.                      (
  Path tmpFile1 = Files.createTempFile("events", ".log");
  FileUtils.printDirEntryInfo(tmpFile1);

  // Create under a specific directory:                                      (
  Path tmpFileDir = Path.of("project");
  Path tmpFile2 = Files.createTempFile(tmpFileDir, "proj_", ".dat", fileAttr);
  Path tmpFile3 = Files.createTempFile(tmpFileDir, "proj_", null,   fileAttr);
  Path tmpFile4 = Files.createTempFile(tmpFileDir, null,    ".dat", fileAttr);
  Path tmpFile5 = Files.createTempFile(tmpFileDir, null,    null,   fileAttr);
  FileUtils.printDirEntryInfo(tmpFile2);
  FileUtils.printDirEntryInfo(tmpFile3);
  FileUtils.printDirEntryInfo(tmpFile4);
  FileUtils.printDirEntryInfo(tmpFile5);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code (*edited to fit on the page*):

Click here to view code image

```
Default temporary directory: /var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/
File:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/events4720093907665196131.log
rw-------
File: project/proj_6062790522710209175.dat
rw-r--r--
File: project/proj_9570151255934538 45.tmp
rw-r--r--
File: project/3032983609251590109.dat
rw-r--r--
File: project/8205471872222375044.tmp
rw-r--r--
```

Creating Symbolic Links

The code below illustrates creating symbolic links to directory entries. We assume that the symbolic link at (1) below does not exist; otherwise, a resounding `File-AlreadyExistsException` will be thrown by the `createSymbolicLink()` method of the `Files` class. However, the target path need *not* exists, but that might limit the file operations that can be performed using the symbolic link. We assume that the target path at (2) exists when the code below is run.

The `createSymbolicLink()` method can be called with a relative path or the absolute path of the target, as shown at (3a) and (3b), respectively. In both cases, the symbolic link will be created with the default file permissions, as we have not specified the optional `FileAttribute` variable arity parameter. Note that the symbolic link is created to a file or a directory, depending on the kind of the target.

The method `readSymbolicLink()` method of the `Files` class returns the path of the target denoted by the symbolic link.

Click here to view code image

```
try {
  Path symbLinkPath  = Path.of(".", "readme_link");                    // (1)
  Path targetPath    = Path.of(".", "project", "backup", "readme.txt"); // (2)

  Path symbLink = Files.createSymbolicLink(symbLinkPath, targetPath);   // (3a
//Path symbLink = Files.createSymbolicLink(symbLinkPath,
//                                         targetPath.toAbsolutePath());// (3b
  Path target = Files.readSymbolicLink(symbLink);                       // (4)

  FileUtils.printDirEntryInfo(symbLink);
  FileUtils.printDirEntryInfo(target);
} catch (FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

Click here to view code image

```
Symbolic link: ./readme_link
rw-r--r--
File: ./project/backup/readme.txt
rw-r--r--
```

Creating Regular and Temporary Directories

The `Files` class provides methods to create regular and temporary directories.

> Click here to view code image
>
> ```
> static Path createDirectory(Path dir, FileAttribute<?>... attrs)
> ```

```
                            throws IOException
```

Creates a new directory denoted by the `Path` object. It does *not* create nonexistent parent directories, and throws a `NoSuchFileException` if any parent directory does not exist. It also throws a `FileAlreadyExistsException` if the directory entry with that name already exists.

Click here to view code image

```
static Path createDirectories(Path dir, FileAttribute<?>... attrs)
            throws IOException
```

Creates a directory by creating all nonexistent parent directories first. It does not throw an exception if any of the directories already exist. If the method fails, some directories may have been created.

Click here to view code image

```
static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
                                throws IOException
static Path createTempDirectory(Path dir, String prefix,
                                FileAttribute<?>... attrs)
                                throws IOException
```

Create a new directory in the *default temporary-file directory* or in the *specified directory* denoted by the `Path` object, using the mandatory non-`null` `prefix` to append to its generated name. A `NoSuchFileException` is thrown by the second method if the specified location does not exist.

For creating directories, we will use the following `FileAttribute` object denoted by the `dirFileAttr` reference (p. 1325). It specifies read, write, and execute permissions for all users: the owner, the group, and others—often called *full permissions*.

Click here to view code image

```
Set<PosixFilePermission> dPerms = PosixFilePermissions.fromString("rwxrwxrwx")
FileAttribute<Set<PosixFilePermission>> dirFileAttr =
        PosixFilePermissions.asFileAttribute(dPerms);
```

Creating Regular Directories

The `Files` class provides two methods to create regular directories. The `create-Directory()` method fails to create a regular directory if the directory already exists, or if the parent directories on the path do not exist. The code below creates a directory with the path `project/bin` relative to the current directory, under the assumption that the directory name `bin` does not exist and the parent directory `./project` exists on the path. The directory is first created with the default file permissions at (1), then deleted at (2) and created again at (3) with specific file permissions. The second call to the `createDirectory()` method at (3) would throw a `FileAlreadyExistsException` if we did not delete the directory before creating it with specific file permissions.

The astute reader will have noticed from the output that the *write* permission for the group and others is not set at creation time by the `createDirectory()` method, regardless of the file permissions specified. This can be remedied by setting the file permissions explicitly after the directory has been created, as at (4).

Click here to view code image

```
try {
  Path regularDir  = Path.of("project", "bin");
  Path createdDir = Files.createDirectory(regularDir);                    // (
  FileUtils.printDirEntryInfo(createdDir);

  if (Files.deleteIfExists(regularDir)) {                                 // (
    System.out.printf("Directory deleted: %s%n", regularDir);
  }
  Path newDir = Files.createDirectory(regularDir, dirFileAttr);           // (
  FileUtils.printDirEntryInfo(newDir);
  Files.setPosixFilePermissions(newDir, dPerms);                          // (
  FileUtils.printDirEntryInfo(newDir);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

Click here to view code image

```
Directory: project/bin
rwxr-xr-
Directory deleted: project/bin
Directory: project/bin
rwxr-xr-x
Directory: project/bin
rwxrwxrwx
```

The method `createDirectories()` first creates all nonexistent parent directories in the path if necessary, and only creates the directory if it does not exist—a convenient way to ensure that a directory always exists. In the code below at (5), the parent directories (in the path `project/branches/maintenance`) are created from top to bottom, relative to the current directory if necessary, and the directory `versions` is only created if it does not exist from before. A `FileAlreadyExistsException` at (6) is thrown if a directory entry named `versions` exists, but is not a directory.

Click here to view code image

```
try {
  Path regularDir2  = Path.of("project", "branches", "maintenance", "versions"
  Path createdDir2 = Files.createDirectories(regularDir2, dirFileAttr);  // (5
  FileUtils.printDirEntryInfo(createdDir2);
} catch (FileAlreadyExistsException fe) {                                // (6
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Directory: project/branches/maintenance/versions
rwxrwxrwx
```

Creating Temporary Directories

Analogous to temporary files, temporary directories can be created by the `create-TempDirectory()` methods of the `Files` class. Again, these directories can be created in the default temporary-file directory, or in a specific location if one is specified in the method call, with either the default file permissions or specific permissions.

The code at (7) creates a temporary directory named `"log_dir"` under the default temporary-file directory. A temporary directory with the same name is also created under a specific location (the `./project` directory) at (8). A `NoSuchFileException` is thrown if the specified location does not exist. A prefix can be specified for the directory name or it can be `null`.

Click here to view code image

```
try  {
  // Create under the default temporary-file directory.                     (7
  Path tmpDirPath1 = Files.createTempDirectory("log_dir", dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath1);

  // Create under a specific location:                                      (8
  Path tmpDirLoc = Path.of("project");
  Path tmpDirPath2 = Files.createTempDirectory(tmpDirLoc, "log_dir", dirFileAt
  Path tmpDirPath3 = Files.createTempDirectory(tmpDirLoc, null, dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath2);
  FileUtils.printDirEntryInfo(tmpDirPath3);

  Files.setPosixFilePermissions(tmpDirPath3, dPerms);                    // (9
  FileUtils.printDirEntryInfo(tmpDirPath3);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code (*edited to fit the page*):

Click here to view code image

```
Directory:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/log_dir18136008118052819366
rwxr-xr-x
Directory: project/log_dir14908011762674796217
rwxr-xr-x
Directory: project/18428782809319921018
rwxr-xr-x
Directory: project/18428782809319921018
rwxrwxrwx
```

We notice from the output that the *write* permission for the group and others is not set at creation time by the `createTempDirectory()` method, regardless of the file permissions specified. Again, this can be remedied by setting the file permissions explicitly after the temporary directory has been created, as at (9).

21.8 Stream Operations on Directory Entries

The `Files` class provides static methods that create specialized streams to implement complex file operations on directory entries, which are concise and powerful, taking full advantage of the Stream API. They are also efficient as the lazy execution of the streams ensures that an element is only made available for processing when needed by the terminal stream operation.

Streams we have seen earlier do not use any system resources, as they are backed by data structures and generator functions. Such streams need not be closed, as they are handled as any other objects by the runtime environment with regard to memory management. In contrast, the streams that are backed by *file system resources* must be closed in order to avoid resource leakage—analogous to using I/O streams. As streams implement the `AutoCloseable` interface, the recommended practice is to use the `try`-with-resources statement, ensuring proper and prompt closing of file system resources after the stream operations complete.

Methods that create streams on directory entries throw a checked

`IOException`, and in particular, they throw a `NoSuchFileException` if the directory entry does not exist. Any code using these methods is forced to handle these exceptions with either a `try-catch-finally` construct or a `throws` clause.

The examples in this section illustrate both closing of streams and handling of exceptions.

Reading Text Lines Using a Functional Stream

The `lines()` method of the `Files` class creates a stream that can be used to read the lines in a text file. It is efficient as it does not read the whole file into memory, making available only one line at a time for processing. Whereas the `BufferedReader` class provides an analogous method that uses a `BufferedReader`, the stream created by the `Files.lines()` method is backed by a more efficient `FileChannel`.

> Click here to view code image
>
> ```
> static Stream<String> lines(Path path) throws IOException
> static Stream<String> lines(Path path, Charset cs) throws IOException
> ```
>
> Return a `Stream` of type `String`, where the elements are text lines read from a file denoted by the specified `path`. The first method decodes the bytes into characters using the UTF-8 charset. The charset to use can be explicitly specified, as in the second method.

As an example of using the `lines()` method, we implement the solution to finding palindromes (§13.3, p. 688), where the words are read from a file.

Click here to view code image

```
Path wordFile = Path.of(".", "project", "wordlist.txt");
System.out.println("Find palindromes, greater than length 2.");
try (Stream<String> stream = Files.lines(wordFile)){
  List<String> palindromes = stream
       .filter(str -> str.length() > 2)
       .filter(str -> str.equals(new StringBuilder(str).reverse().toString()))
       .toList();
  System.out.printf("List of palindromes:   %s%n", palindromes);
  System.out.printf("Number of palindromes: %s%n", palindromes.size());
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code (*edited to fit on the page*):

Click here to view code image

```
Find palindromes, greater than length 2.
List of palindromes:   [aba, abba, aga, aha, ...]
Number of palindromes: 90
```

The following code creates a map to count the number of lines with different lengths:

Click here to view code image

```
Path textFile = Path.of(".", "project", "linesOnly.txt");
try (Stream<String> stream = Files.lines(textFile)) {
  Map<Integer, Long> grpMap =
      stream.collect(Collectors.groupingBy(String::length,
                                           Collectors.counting()));
  System.out.println(grpMap);
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code:

```
{4=1, 21=2, 10=1, 12=1}
```

Directory Hierarchy Traversal

Tree traversal is an important topic in computer science. Other synonyms used in the literature for traversing such structures are *walking*, *navigating*, and *visiting* a tree.

Note that in order to traverse a directory hierarchy and access its entries, the directory must have *execute* permission.

The `Files` class provides methods to traverse a directory hierarchy which has a tree structure. We will use the directory hierarchy shown below to illustrate these traversal methods. The directory hierarchy below is rooted at directory `a`—that is, this directory is the *start* or the *root* of the directory hierarchy. Traversal of the directory hierarchy starts by visiting directory `a` and is considered to be at *depth* 0. Dropping down each level of directories in the hierarchy increases the depth. The directory `a/b` and the file `a/x.txt` are at depth 1 in relation to the directory `a`, and are said to be *siblings* at depth 1. The directory hierarchy below has a maximum depth of 3, as there are no directories below this depth. Note that the file `a/b/c/dir_link` is a symbolic link to the directory `a/b`, and as we shall see, it can influence the traversal if symbolic links are to be followed.

Traversal Strategies

There are primarily two main strategies for traversing a tree structure, or as in our case, a directory hierarchy. Both strategies start at the root of the directory hierarchy.

- *Depth-first traversal*

  Any subdirectory encountered at a particular depth level is traversed to its maximum depth before continuing with any sibling of this subdirectory at this depth level. The depth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how at depth level 2, the directory `a/b/c` is traversed completely before traversing its sibling directory `a/b/d`. The traversal methods of the `Files` class use the depth-first traversal strategy.

  Click here to view code image

  ```
  Visited entry          Depth
  ./a                    0
  ./a/x.txt              1
  ./a/b                  1
  ./a/b/c                2
  ./a/b/c/z.txt          3
  ./a/b/c/dir_link       3
  ./a/b/d                2
  ./a/b/d/y.txt          3
  ```

- *Breadth-first traversal*

  The siblings at each depth level are traversed before moving on to the next depth level—in other words, the traversal is by depth level. The breadth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how entries at each depth are traversed before traversing the entries at the next depth level.

  Click here to view code image

  ```
  Visited entry          Depth
  ./a                    0
  ./a/x.txt              1
  ./a/b                  1
  ./a/b/d                2
  ./a/b/c                2
  ./a/b/c/z.txt          3
  ./a/b/c/dir_link       3
  ./a/b/d/y.txt          3
  ```

Both strategies have their pros and cons, and each excels at solving particular traversal problems. For example, finding a search goal that is closest to the root is best found by the breadth-first strategy in the shortest amount of time, whereas a goal that is farthest from the root is best found by the depth-first strategy in the shortest amount of time. Depth-first search may need to backtrack when trying to find a solution, whereas breadth-first search is more predictive. Breath-first search requires more memory as entries at previous levels must be maintained. The interested reader should consult the ample body of literature readily available on this important subject of trees and graph algorithms.

Listing the Immediate Entries in a Directory

A common file operation is to list the entries in a directory, similar to basic usage of the DOS command `dir` or the Unix command `ls`. The `list()` method of the `Files` class creates a stream whose elements are `Path` objects that denote the entries in a directory.

> Click here to view code image
>
> ```
> static Stream<Path> list(Path dir) throws IOException
> ```
>
> Creates a lazily populated `Stream` whose elements are the entries in the directory—it does *not* recurse on the entries of the directory. The stream does not include the directory itself or its parent. It throws a `java.nio.file.NotDirectoryException` if it is not a directory (*optional specific exception*). Also, it does not follow symbolic links.

The following code lists the entries under the directory `a`. Note that the stream only contains the *immediate* entries in the directory. Any subdirectories under the specified directory are *not* traversed. The `Path` object passed a parameter must denote a directory, or a disgruntled `NotDirectoryException` is thrown.

Click here to view code image

```
Path dir = Path.of(".", "a");
System.out.printf("Immediate entries under directory \"%s\":%n", dir);
try(Stream<Path> stream = Files.list(dir)) {
  stream.forEach(System.out::println);
} catch (NotDirectoryException nde) {
nde.printStackTrace();
} catch (IOException ioe) {
ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Immediate entries under directory "./a":
./a/x.txt
./a/b
```

File Visit Option

In the file operations we have seen so far, the default behavior was to follow symbolic links. Not so in the case of the traversal methods `find()` and `walk()` of the `Files` class—these methods do *not* follow symbolic links, unless instructed explicitly to do so. The `FileVisitOption` enum type defines the constant `FOLLOW_LINKS` (Table 21.13) that can be specified to indicate that symbolic links should be followed when using these methods.

Table 21.13 *File Visit Option*

Enum java.nio.file.FileVisitOp	**Description**
`FOLLOW_LINKS`	Indicates that symbolic links should be followed.

Walking the Directory Hierarchy

The `walk()` method of the `Files` class creates a stream to walk or traverse the directory hierarchy rooted at a specific directory entry.

> Click here to view code image
>
> ```
> static Stream<Path> walk(Path start, FileVisitOption... options)
>                          throws IOException
> static Stream<Path> walk(Path start, int depth,
>                          FileVisitOption... options)
>                          throws IOException
> ```
>
> Return a `Stream` that is lazily populated with `Path` objects by walking the directory hierarchy rooted at the entry specified by the `start` parameter. The `start` parameter can be a directory or a file.
>
> The first method is equivalent to calling the second method with a depth of `Integer.MAX_VALUE`. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the maximum depth of the directory hierarchy rooted at the `start` entry and any depth that is specified or implied.
>
> These methods do *not* follow symbolic links, unless the constant `FileVisitOption.FOLLOW_LINKS` is specified.

Example 21.5 illustrates using the `walk()` method. The hierarchy of directory `a` (p. 1347) is traversed to illustrate different scenarios. The code at (1) creates the symbolic link `a/b/c/dir_link` to the directory `a/b`, if necessary.

Example 21.5 *Traversing the Directory Hierarchy*

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class WalkTheWalk {

  public static void main(String[] args) throws IOException {

    // Creating symbolic link.                                          // (1)
    try {
      Path targetPath = Path.of(".", "a", "b");
      Path symbLinkPath  = Path.of(".", "a", "b", "c", "dir_link");
      if (Files.notExists(symbLinkPath, LinkOption.NOFOLLOW_LINKS)) {
        Files.createSymbolicLink(symbLinkPath, targetPath.toAbsolutePath());
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return;
    }

    // Do the walk.                                                      // (2)
    Path start = Path.of(".", "a");
    int MAX_DEPTH = 4;

    for (int depth = 0; depth <= MAX_DEPTH; ++depth)  {                  // (3)
      try(Stream<Path> stream = Files.walk(start, depth,                 // (4)
                                           FileVisitOption.FOLLOW_LINKS)) {
        System.out.println("Depth limit: " + depth);
        stream.forEach(System.out::println);
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Depth: 0
./a
Depth: 1
./a
./a/x.txt
./a/b
Depth: 2
./a
./a/x.txt
./a/b
./a/b/c
```

```
./a/b/d
Depth: 3
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
./a/b/c/dir_link
./a/b/d
./a/b/d/y.txt
Depth: 4
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
Exception in thread "main" java.io.UncheckedIOException:
    java.nio.file.FileSystemLoopException: ./a/b/c/dir_link
...
```

Limiting Traversal by Depth Specification

The `walk()` method at (4) in Example 21.5 is called multiple times in the `for(;;)` loop at (3). From the output, we can see that, for each value of the loop variable `depth`, the `walk()` method starts the traversal at the directory `a` and descends to the depth level given by the loop variable `depth`. For example, when the value of the `depth` variable is `3`, traversal descends from depth level 0 to depth level 3.

If the constant `FileVisitOption.FOLLOW_LINKS` is *not* specified, so that symbolic links are not followed, the traversal will only go as far as the minimum of the maximum depth of the hierarchy and any specified depth level.

Specifying the depth level can result in a more efficient search if it is known that the result is at an approximate depth level, thus avoiding useless searching farther down in the hierarchy.

Handling Symbolic Links

Specifying the constant `FileVisitOption.FOLLOW_LINKS` in the `walk()` method call results in symbolic links being followed.

A symbolic link to a file is not a problem, as after visiting the target file, the traversal can continue with the next sibling of the symbolic link.

A symbolic link to a directory that is *not* a *parent directory* of the symbolic link is also not a problem, as traversal can continue with the next sibling of the symbolic link, after visiting the target directory.

The problem arises when a symbolic link is to a directory that is a *parent directory* of the symbolic link. That creates a *cyclic path* *dependency*. An example of such a cyclic path dependency is introduced by the symbolic link `a/b/c/dir_link` to one of its parent directories, `a/b`, as can be seen in the figure of the directory hierarchy (p. 1347).

When the constant `FileVisitOption.FOLLOW_LINKS` is specified in the `walk()` method call, the method detects such cyclic path dependencies by monitoring which entries have been visited. Example 21.5 illustrates this scenario when the constant `FileVisitOption.FOLLOW_LINKS` is specified and the specified depth is greater than 3. From the output, we can see that, at depth level 4, the method detects that the symbolic link `a/b/c/dir_link` to its parent directory `a/b` creates a cyclic path dependency, as the target directory lies on the path to the symbolic link and has been visited before. The method throws a hefty `FileSystemLoopException` to announce the outcome.

Copying an Entire Directory

The `Files.copy()` method only copies a single file or creates an empty directory at its destination, depending on whether the source is a file or a directory, respectively. Shown below is the `copyEntireDirectory()` method that copies an *entire* directory.

Click here to view code image

```
/** Declared in FileUtils class.
 * Copy an entire directory.
 * @param sourceDir        Directory to copy.
 * @param destinationDir   Directory to which the source directory is copied.
 * @param options          Copy options for all entries.
 */
public static void copyEntireDirectory(Path sourceDir,
                                       Path destinationDir,
                                       CopyOption... options)  {
  try (Stream<Path> stream = Files.walk(sourceDir)) {                   // (1
    stream.forEach(entry -> {
      Path relativeEntryPath = sourceDir.relativize(entry);            // (2
      Path destination  = destinationDir.resolve(relativeEntryPath);   // (3
      try {
        Files.copy(entry, destination, options);                       // (4
      } catch (DirectoryNotEmptyException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    });
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

The method `copyEntireDirectory()` copies each entry with the

`Files.copy()` method at (4) as the directory hierarchy is traversed in the stream created by the `Files.walk()` method at (1). For each entry, the destination where the entry should be copied is determined by the code at (2) and (3).

First, the *relative* path between the source directory path and the current entry path is determined by the `relativize()` method at (2):

Click here to view code image

```
Source directory:      ./a/b
Current entry:         ./a/b/c/d
Relative entry path:   c/d
```

The *destination* path to copy the current entry is determined by joining the destination directory path with the relative entry path by calling the `resolve()` method at (3):

Click here to view code image

```
Destination directory:   ./x/y
Relative entry path:     c/d
Destination entry paths: ./x/y/c/d
```

In the scenario above, the entry `./a/b/c/d` is copied to the destination path `./x/y/c/ d` during the copying of source directory `./a/b` to the destination directory `./x/y`.

Click here to view code image

```
CopyOption[] options = new CopyOption[] {
    StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES,
    LinkOption.NOFOLLOW_LINKS};
Path sourceDirectory      = Path.of(".", "a", "b");        //      ./a/b
Path destinationDirectory = Path.of(".", "x", "y");        // (5a) ./x/y
// Path destinationDirectory = Path.of(".", "x")
//              .resolve(sourceDirectory.getFileName());   // (5b) ./x/b
FileUtils.copyEntireDirectory(sourceDirectory, destinationDirectory, options);
```

If the destination directory should have the same name as the source directory, we can use the code at (5b) rather than the code at (5a).

A few things should be noted about the declaration of the `copyEntireDirectory()` method. It can be customized to copy the entries by specifying copy options. It closes the stream created by the `walk()` method in a `try`-with-resources statement. Calling the method `copy()` in the body of the lambda expression requires handling any `IOException` inside the lambda body. Any `IOException` from calling the `walk()` method is also explicitly handled.

Searching for Directory Entries

The `find()` method of the `Files` class can be used for searching or finding directory entries in the file system.

> Click here to view code image
>
> ```
> static Stream<Path> find(Path start, int depth,
>                          BiPredicate<Path,BasicFileAttributes> matcher,
>                          FileVisitOption... options) throws IOException
> ```
>
> Returns a `Stream` that is lazily populated with `Path` objects by searching for entries in a directory hierarchy rooted at the `Path` object denoted by the `start` parameter. The `start` parameter can be a directory or a file.
>
> The method walks the directory hierarchy in exactly the same manner as the `walk()` method. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the actual depth of the directory hierarchy rooted at the `start` entry and the `depth` that is specified.
>
> The `matcher` parameter defines a `BiPredicate` that is used to decide whether a directory entry should be included in the stream. For each directory entry in the stream, the `BiPredicate` is invoked on its `Path` and its `BasicFileAttributes`— making it possible to define a customized filter.
>
> This method does *not* follow symbolic links, unless the constant `FileVisitOption.FOLLOW_LINKS` is specified.

The `find()` method is analogous to the `walk()` method in many respects: It traverses the directory hierarchy in the same way, does not follow symbolic links by default, follows symbolic links only if the constant `FileVisitOption.FOLLOW_LINKS` is specified, and monitors visiting entries to detect cyclic path dependencies since the depth limit is always specified.

Whereas both `walk()` and `find()` methods create a stream of `Path` objects, the `find()` method also allows a matcher for the search to be defined by a lambda expression that implements the `BiPredicate<Path, BasicFileAttributes>` functional interface. This matcher is applied by the method and determines whether an entry is allowed in the stream. This is in contrast to an explicit intermediate filter operation on the stream, as in the case of a stream created by the `walk()` method. The `find()` method supplies the `BasicFileAttributes` object associated with a `Path`—analogous to using the `readAttributes()` method of the `Files` class (p. 1328). Given the `Path` object and its associated `BasicFileAttributes` object with the basic set of read-only file attributes, it is possible to write complex search criteria on a directory entry. The methods of the `BasicFileAttributes` interface also do not throw checked exceptions, and are therefore convenient to call in a lambda expression, as opposed to corresponding methods of the `Files` class.

In the following code, we use the `find()` method to find regular files whose name ends with `".txt"` and whose size is greater than 0 bytes. The `BiPredicate` at (1) defines the filtering criteria based on querying the `Path` object in the stream for its file extension and its associated `BasicFileAttributes` object with read-only file attributes for whether it is a regular file and for its size.

Click here to view code image

```
System.out.println("Find regular files whose name ends with \".txt\""
                 + " and whose size is > 0:");
Path startEntry = Path.of(".", "a");
int depth = 5;
try (var pStream = Files.find(startEntry, depth,
                              (path, attrs) -> attrs.isRegularFile()        //
                              && attrs.size() > 0
                              && path.toString().endsWith(".txt"))) {
   List<Path> pList = pStream.toList();
   System.out.println(pList);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Find regular files whose name ends with ".txt" and whose size is > 0:
[./a/x.txt, ./a/b/c/z.txt, ./a/b/d/y.txt]
```

Note that the method `find()` requires the depth of the search to be specified. However, the actual depth traversed is always the minimum of the maximum depth of the directory hierarchy and the depth specified. The maximum depth of the hierarchy rooted at directory `a` is 3 and the specified depth is 5. The actual depth traversed in the directory hierarchy is thus 3. In the code above, different values for the depth can give different results.

If the constant `FileVisitOption.FOLLOW_LINKS` is specified in the `find()` method, its behavior is analogous to the behavior of the `walk()` method. It will keep track of the directories visited, and any cyclic path dependency encountered will unceremoniously result in a `FileSystemLoopException`. The curious reader is encouraged to experiment with the code above by specifying the constant `FileVisitOption.FOLLOW_LINKS` and passing different values for the depth in the `find()` method call.

Review Questions

21.1 Assume that the current directory has the absolute path `/wrk`.

Which of the following statements are true about the following program?

Click here to view code image

```
import java.io.IOException;
import java.nio.file.Path;

public final class Filing {
  public static void main (String[] args) throws IOException {
    Path file = Path.of("./document", "../book/../chapter1");
    System.out.println(file.toAbsolutePath());    // (1)
    System.out.println(file.toRealPath());        // (2)
    System.out.println(file.normalize());         // (3)
    System.out.println(file.toString());          // (4)
    System.out.println(file.getParent());         // (5)
  }
}
```

Select the two correct answers.

a. The line at (1) will print `/wrk/document/book/chapter1`.

b. The line at (2) will print `/wrk/chapter1`.

c. The line at (3) will print `chapter1`.

d. The line at (4) will print `./document/book/chapter1`.

e. The line at (5) will print `./document/book/`.

21.2 Given the following program:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class ListingFiles {
  public static void main(String[] args) throws IOException {
    Path dirPath = Path.of(".", "wrk");
    printFiles1(dirPath);
    printFiles2(dirPath);
    printFiles3(dirPath);
  }
  public static void printFiles1(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.list(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles2(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.walk(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles3(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.find(dirPath, Integer.MAX_VALUE,
                                         (p, a) -> true)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }
}
```

Assume that the directory `./wrk` has the following directory hierarchy:

```
./wrk
    src
       readme.txt
```

Which statement is true about the program?
Select the one correct answer.

a. All three methods `printFiles1()`, `printFiles2()`, and `printFiles3()` will produce the same output.

b. Only the methods `printFiles1()` and `printFiles2()` will produce the same output.

c. Only the methods `printFiles2()` and `printFiles3()` will produce the same output.

d. Only the methods `printFiles1()` and `printFiles3()` will produce the same output.

e. The program will fail to compile because the `list()` method does not exist in the `Files` class.

21.3 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ1 {
  public static void main(String[] args) {

    Path earth = Path.of("/", "planets", "earth");
    Path moonOrbit = earth.resolve(Path.of("moon", "orbit.param"));
    Path mars = earth.resolveSibling("mars");
    Path fromMarsToMoon = mars.relativize(moonOrbit);
    System.out.println(fromMarsToMoon);
  }
}
```

What is the result?

Select the one correct answer.

- **a.** `../earth/moon/orbit.param`
- **b.** `/planets/mars/../earth/moon/orbit.param`
- **c.** `/planets/earth/moon/orbit.param`
- **d.** `./mars/../earth/moon/orbit.param`
- **e.** The program will throw an exception at runtime.

21.4 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ2 {
  public static void main(String[] args) {
    /* Assume current directory path is /planets. */
    Path path = Path.of("./mars/../earth").normalize().toAbsolutePath();
    System.out.println(path.getNameCount());
  }
}
```

What is the result?

Select the one correct answer.

- **a.** `1`
- **b.** `2`
- **c.** `5`
- **d.** The program will throw an exception at runtime.

21.5 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ3 {
  public static void main(String[] args) {
      try (var stream = Files.list(Path.of("/test")
                                       .toRealPath(LinkOption.NOFOLLOW_L
        stream.filter(p -> p.getFileName().toString().endsWith("txt"))
              .forEach(p -> System.out.println(p));
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

and the following hierarchy for the `/test` directory:

```
/test
    a.txt
        b.txt
    c
        d.txt
    e.txt
    f.txt
```

What is the result?

Select the one correct answer.

a.

```
/test/a.txt
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

b.

```
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

c.

```
/test/a.txt/b.txt
/test/e.txt
/test/f.txt
```

d.

```
/test/a.txt
/test/e.txt
/test/f.txt
```

e.

```
a.txt/b.txt
c/d.txt
e.txt
f.txt
```

f.

```
b.txt
d.txt
e.txt
f.txt
```

g.

```
a.txt/b.txt
e.txt
f.txt
```

h.

```
a.txt
b.txt
d.txt
e.txt
f.txt
```

21.6 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ4 {
  public static void main(String[] args) {

    try (var stream = Files.walk(Path.of("/test")
                                      .toRealPath(LinkOption.NOFOLLOW_LIN
      stream.map(p -> p.getFileName().toString())
            .filter(p -> p.endsWith("txt"))
            .sorted()
            .forEach(System.out::println);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

and the following hierarchy for the `/test` directory:

```
/test
    a.txt
        b.txt
    c
        d.txt
```

```
e.txt
f.txt
```

What is the result?

Select the one correct answer.

a.
```
/test/a.txt
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

b.
```
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

c.
```
/test/a.txt/b.txt
/test/e.txt
/test/f.txt
```

d.

```
/test/a.txt
/test/e.txt
/test/f.txt
```

e.
```
b.txt
d.txt
e.txt
f.txt
```

f.
```
b.txt
e.txt
f.txt
```

g.
```
a.txt
b.txt
d.txt
e.txt
f.txt
```

21.7 Which of the following statements is true?

Select the one correct answer.

a. The `Files.createDirectories()` method throws an exception when the directory to create already exists.

b. The `Files.delete()` method throws an exception when trying to delete a nonexistent path.

c. The `Files.move()` method throws an exception when attempting to move a non-empty directory in the same file system.

d. The `Files.exists()` method throws an exception when a path does not exist in the file system.

21.8 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ6 {
  public static void main(String[] args) {

    try {
      Path p1 = Path.of("/users/joe/test/a.jpg");
      Path p2 = Path.of("/users").resolve(p1.getName(1).resolve("test/a.
      Files.move(p1, p2);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Assuming that the relevant paths exist, what is the result?

Select the one correct answer.

a. The file `a.jpg` is moved to the `/users` directory.

b. The file `a.jpg` is moved to the `/users/joe/test` directory.

c. No action is taken when trying to move the file `a.jpg`.

d. An `IOException` is thrown when trying to move the file `a.jpg`.

21.9 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ7 {
  public static void main(String[] args) {
    Path p1 = Path.of("/test");
    Path p2 = Path.of("store");
    System.out.println(p1.resolve(p2));
    System.out.println(p2.resolve(p1));
  }
}
```

What is the result?

Select the one correct answer.

a.
```
/test/store
/store/test
```

b.
```
/test
/store
```

c.
```
/test/store
/test
```

d.
```
/test/store
/test/store
```

21.10 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
public class RQ8 {
  public static void main(String[] args) {
    try {
      Path source = Path.of("/test/readme.txt");
      Path destination = Path.of("/backup/readme_save.txt");
      Files.copy(source, destination);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Assuming that both source and destination files exist and are accessible, what is the result?

Select the one correct answer.

a. The destination file will be overwritten with the content of the source file.

b. The content of the source file will be appended to the end of the destination file.

c. The content of the source file will be inserted at the beginning of the destination file.

d. The program will throw an exception at runtime.

21.11 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
public class RQ9 {
  public static void main(String[] args) {
    Path p = Path.of("/test/test.xhtml");
    try (var stream = Files.lines(p)){
      String result = stream.filter(s->s.startsWith("<"))
                            .map(s->s.substring(s.indexOf(">"), s.indexO
                            .reduce("", (s1,s2) -> s1+s2);
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

and that the `/test/test.xhtml` file contains the following lines:

```
<!DOCTYPE html>
<html>
<body>
Hello World
</body>
</html>
```

What is the result?

Select the one correct answer.

a. An empty string

b. < < < < <

c. < > < > < > < > < >

d. >>>>>

e. The program will throw an exception at runtime.

21.12 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

public class RQ10 {
  public static void main(String[] args) {
    Path directory = Path.of("/test/data");
    Path file = Path.of("/test/data/info.txt");
    try {
      Set<PosixFilePermission> permissions = new HashSet<>();
      permissions.add(PosixFilePermission.OWNER_READ);
      Files.setPosixFilePermissions(directory, permissions);

      permissions.add(PosixFilePermission.OWNER_WRITE);
      Files.setPosixFilePermissions(file, permissions);
      try (var stream = Files.walk(directory)) {
        stream.forEach(System.out::println);
      }
    } catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

assume that the directory `/test/data` exists and only contains the `info.txt` file, and that full permissions are set for both the directory and the file prior to program execution.

What is the result?

Select the one correct answer.

a. `/test/data`

b. `/test/data/info.txt`

c. Click here to view code image

```
java.nio.file.AccessDeniedException:/test/data/info.txt
```

d. Click here to view code image

```
java.nio.file.AccessDeniedException:/test/data
```

21.13 Given the following code:

Click here to view code image

```
import static java.nio.file.attribute.PosixFilePermission.*;
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.Set;

public class RQ11 {
  public static void main(String[] args) {
    try {
      Path file = Path.of("/test/data/info.txt");
      Set<PosixFilePermission> perms
              = PosixFilePermissions.fromString("---------");
      Files.setPosixFilePermissions(file, perms);
      PosixFileAttributeView pfaView = Files.getFileAttributeView(file,
          PosixFileAttributeView.class);
      PosixFileAttributes pfa = pfaView.readAttributes();

      perms = Set.of(OWNER_READ, GROUP_WRITE, OTHERS_READ);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.add(OWNER_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.remove(GROUP_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      System.out.println(perms);

    } catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

What is the result?

Select the one correct answer.

a. Click here to view code image

```
[OWNER_READ, OWNER_WRITE, GROUP_WRITE, OTHERS_READ]
```

b. Click here to view code image

```
[OWNER_READ, OWNER_WRITE, OTHERS_READ]
```

c. `[OWNER_WRITE]`

d. `[]`

e. The program will throw an exception at runtime.

21.14 Given the following code:

Click here to view code image

```
import java.io.*;
import java.net.*;
import java.nio.file.*;
public class RQ12 {
  public static void main(String[] args) {
    try {
      URI uri = URI.create("file:///test/ora.xhtml");
      Path p1 = Path.of(uri);
      Path p2 = Path.of("/test/ora.xhtml");
      Files.copy(p1, p2);
      File file = p2.toFile();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Assume that the file `/test/ora.xhtml` exists. What is the result?

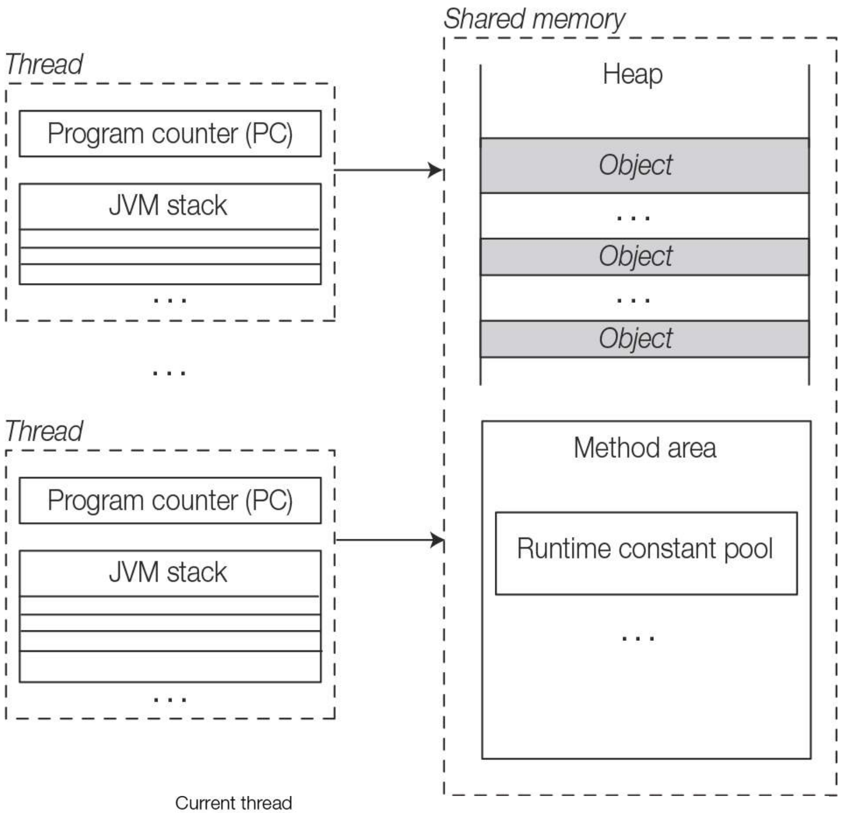
Select the one correct answer.

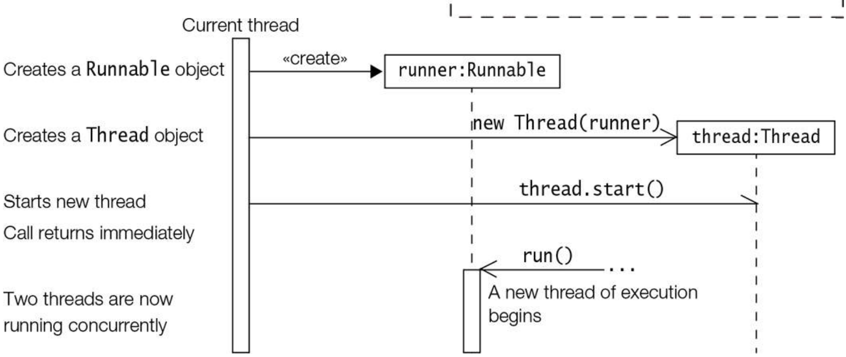
a. No output is produced.

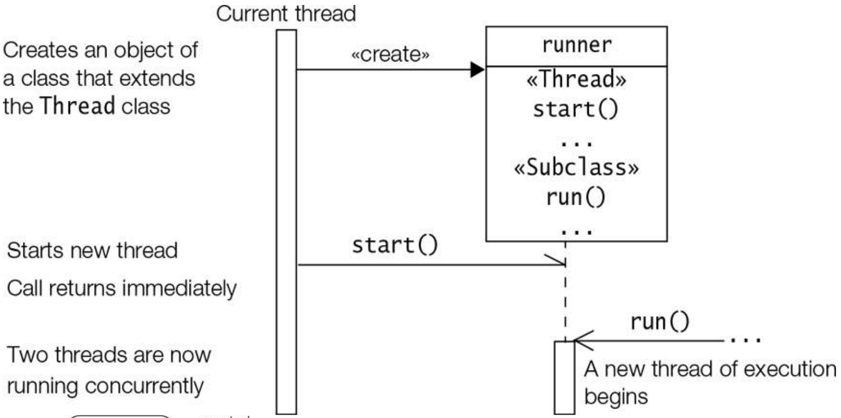
b. An exception is thrown when converting a `URI` object to a `Path` object.

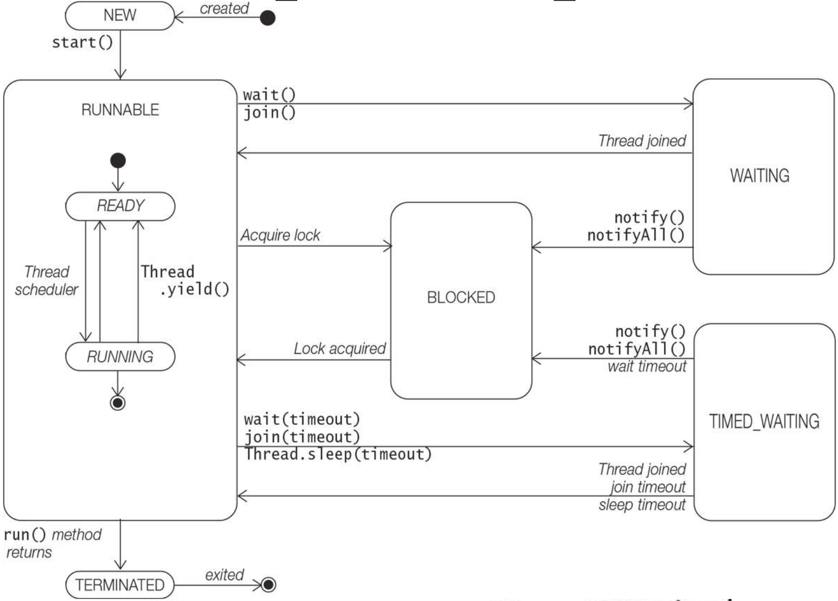
c. An exception is thrown when copying path `p1` to path `p2`.

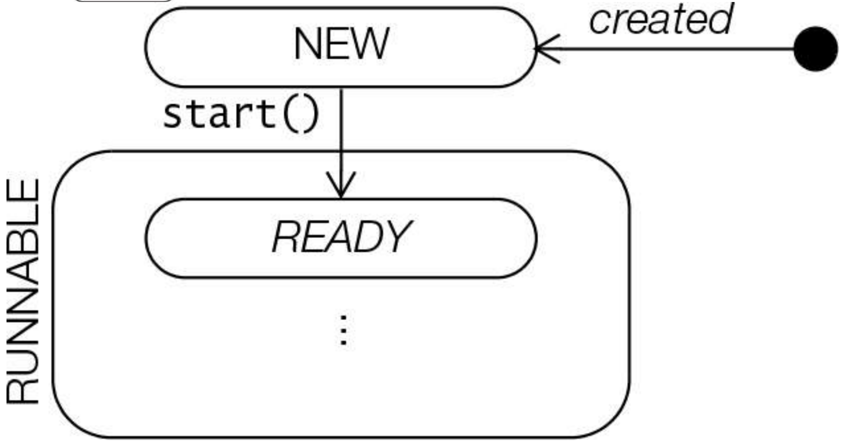
d. An exception is thrown when converting a `Path` object to a `File` object.

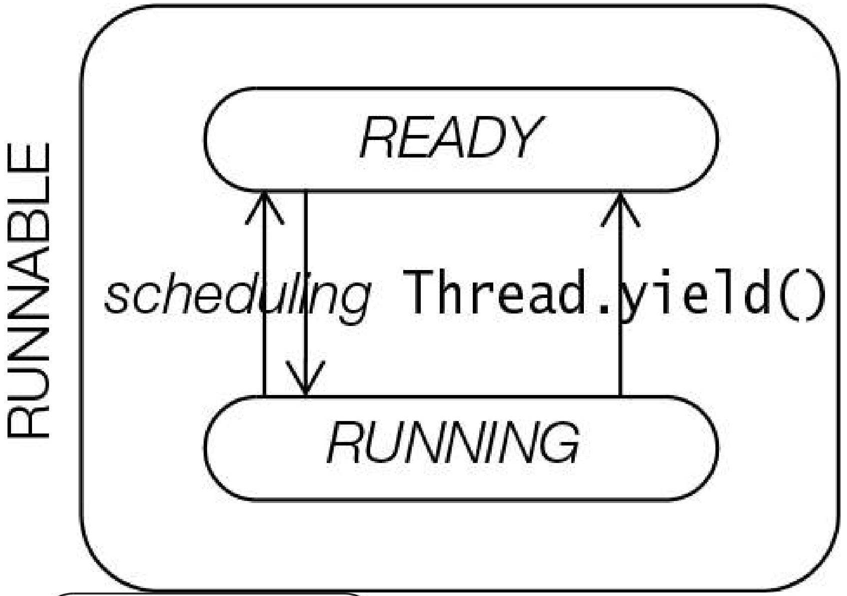

Shared memory
Thread
Program counter (PC)
JVM stack
. . .
. . .
Thread
Program counter (PC)
JVM stack
. . .
Heap
Object
. . .
Object
. . .
Object
Method area
Runtime constant pool
. . .

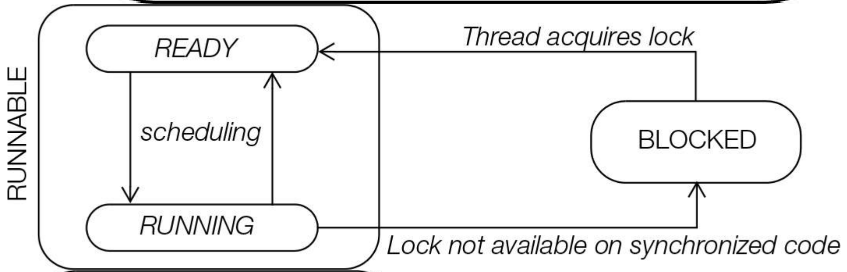

Current thread
Creates a Runnable object
«create»
runner:Runnable
Creates a Thread object
new Thread(runner)
thread:Thread
Starts new thread
Call returns immediately
thread.start()
run()
. . .
A new thread of execution begins
Two threads are now running concurrently

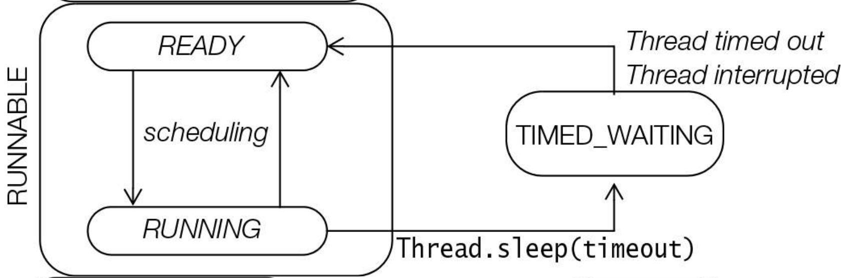

Current thread
Creates an object of a class that extends the Thread class
«create»
runner
«Thread»
start()
...
«Subclass»
run()
...
Starts new thread
Call returns immediately
start()
run()
...
Two threads are now running concurrently
A new thread of execution begins

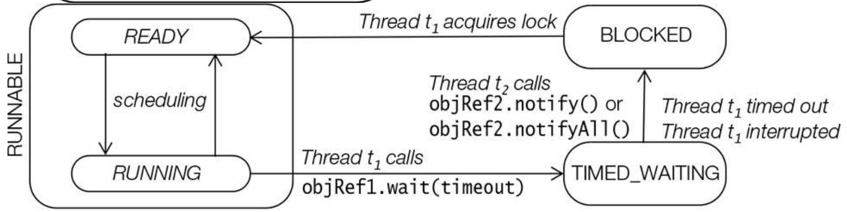

created
NEW
start()
RUNNABLE
READY
Thread scheduler
Thread
.yield()
RUNNING
wait()
join()
Thread joined
WAITING
Acquire lock
BLOCKED
notify()
notifyAll()
Lock acquired
notify()
notifyAll()
wait timeout
wait(timeout)
join(timeout)
Thread.sleep(timeout)
TIMED_WAITING
Thread joined
join timeout
sleep timeout
run() method returns
TERMINATED
exited


NEW
created
start()
RUNNABLE
READY
⋮

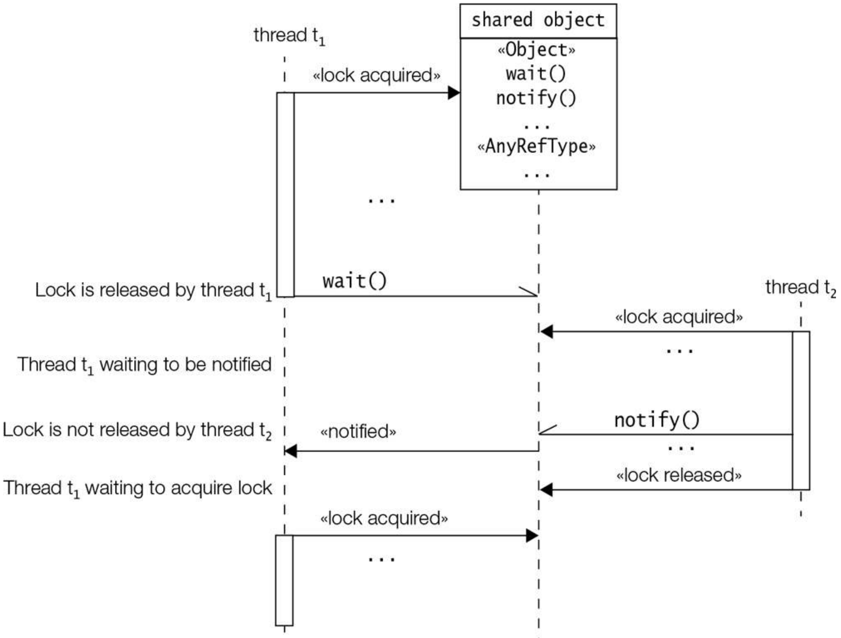

RUNNABLE
READY
scheduling
Thread.yield()
RUNNING

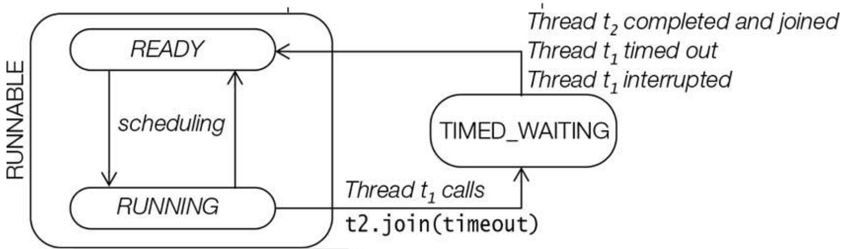

RUNNABLE
READY
scheduling
RUNNING
Thread acquires lock
BLOCKED
Lock not available on synchronized code

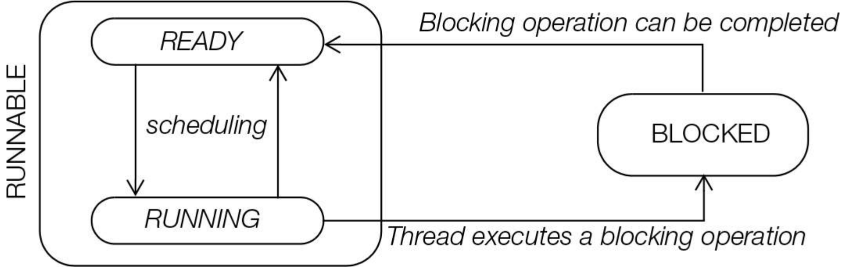

RUNNABLE
READY
scheduling
RUNNING
Thread timed out
Thread interrupted
TIMED_WAITING
Thread.sleep(timeout)

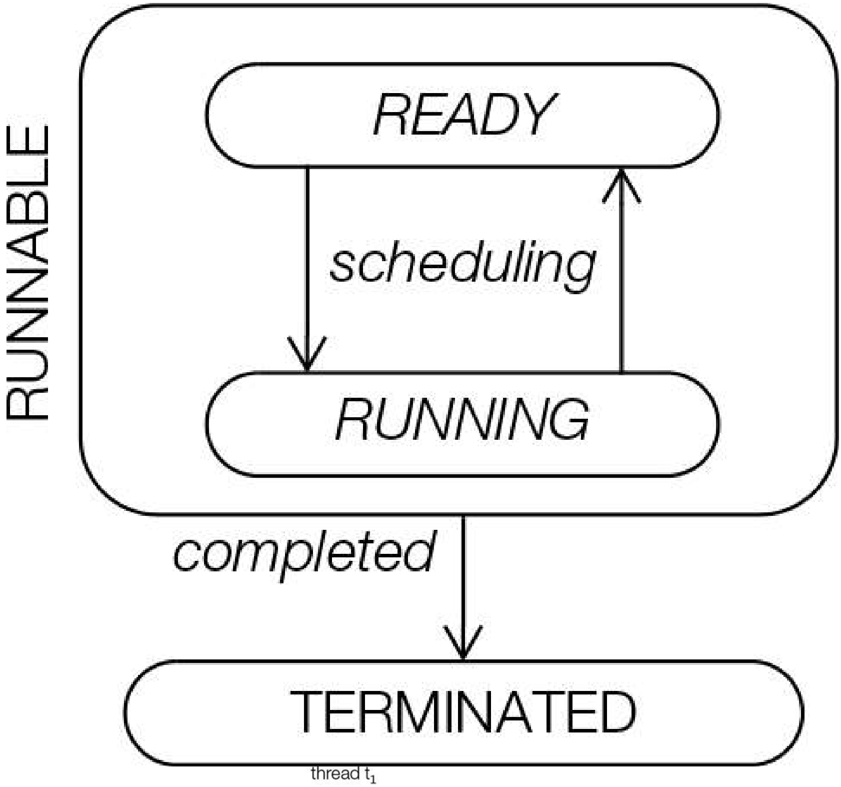

RUNNABLE
READY
scheduling
RUNNING
Thread t1 acquires lock
BLOCKED
Thread t2 calls
objRef2.notify() or
objRef2.notifyAll()
Thread t1 timed out
Thread t1 interrupted
Thread t1 calls
objRef1.wait(timeout)
TIMED_WAITING

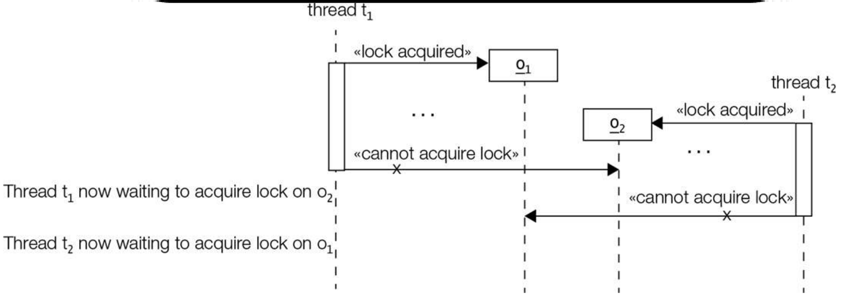
`objRef1` and `objRef2` are aliases


thread t_1
shared object
«Object»
wait()
notify()
...
«AnyRefType»
...
«lock acquired»
...
wait()
Lock is released by thread t_1
thread t_2
«lock acquired»
...
Thread t_1 waiting to be notified
notify()
Lock is not released by thread t_2
«notified»
...
«lock released»
Thread t_1 waiting to acquire lock
«lock acquired»
...


RUNNABLE
READY
scheduling
RUNNING
Thread t_2 completed and joined
Thread t_1 timed out
Thread t_1 interrupted
TIMED_WAITING
Thread t_1 calls
t2.join(timeout)


RUNNABLE
READY
scheduling
RUNNING
Blocking operation can be completed
BLOCKED
Thread executes a blocking operation


RUNNABLE
READY
scheduling
RUNNING
completed
TERMINATED


thread t_1
«lock acquired»
o_1
thread t_2
«lock acquired»
o_2
. . .
. . .
«cannot acquire lock»
Thread t_1 now waiting to acquire lock on o_2
«cannot acquire lock»
Thread t_2 now waiting to acquire lock on o_1

Concurrency: Part I 22

Chapter Topics

- Distinguishing between concurrency and parallelism
- Understanding runtime organization for thread execution: what is shared memory and what is private to a thread
- Creating threads by extending the `Thread` class or by implementing the `Runnable` interface
- Writing synchronized code using `synchronized` methods and `synchronized` statements to achieve mutually exclusive access to shared resources
- Managing the thread lifecycle: thread states, the transitions between them, and thread coordination
- Understanding threading problems: liveness, fairness, deadlock, starvation, livelock, and memory consistency errors
- Understanding which thread behavior a program can take as guaranteed, and which behavior is not guaranteed, including the happens-before relationship

Java SE 17 Developer Exam Ob

[8.1] Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different Executor services and concurrent API §22.1, p. 1367, to §22.5, p. 1408

- *Creating threads with the* `Runnable` *functional interface and managing the thread lifecycle are covered in this chapter.*
- *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in §23.2, p. 1423.*

Java SE 11 Developer Exam Ob

[8.1] Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API §22.3, p. 1371

- *Creating threads with the* `Runnable` *functional interface is covered in this chapter.*
- *Creating worker threads with the* `Callable` *functional interface, and managing concurrency using executor services of the Concurrency API are covered in §23.2, p. 1423.*

Support for multithreaded programming (also called *concurrent programming*) is an integral feature of the Java ecosystem to harness the computing power of multiprocessor and multicore architectures.

The programming language, the tools (particularly the JVM), and the APIs have all evolved to meet this challenge.

This chapter focuses on *low-level* support for multithreaded programming. This includes creating threads to execute tasks, understanding the thread lifecycle, how threads cooperate, and thread issues that are common in concurrent programming.

Chapter 23, p. 1419, deals with *high-level* support for building massively concurrent applications. This support is primarily provided by the frameworks in the `java.util.concurrent` package. High-level mechanisms allow the programmer to concentrate on *task management* in concurrent applications, while taking care of low-level *thread management*.

22.1 Threads and Concurrency

We first look at some basic concepts before diving into multithreaded programming in Java.

Multitasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking (associated synonymously with *concurrency* in Java)

At the coarse-grain level there is *process-based* multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running a spreadsheet program while also working with a word processor. At the fine-grain level there is *thread-based* multitasking, which allows parts of the *same* program to run concurrently on the computer. A familiar example is a word processor that is formatting text as it is typed and is spell-checking at the same time. This is only feasible if the two tasks are performed by two independent paths of execution at runtime. The two tasks would correspond to executing parts of the program code concurrently.

The sequence of code executed for each task defines an *independent sequential path of execution* within the program, and is called a *thread* (*of execution*). Many threads can run concurrently within a program, doing different tasks. At runtime, threads in a program exist in a common memory space, and can therefore share both data and code (i.e., they are *lightweight* compared to processes). They also share the process running the program.

In a single-threaded environment only one task at a time can be performed. CPU cycles are wasted—for example, when waiting for user input. Multitasking allows idle CPU time to be put to good use.

Some advantages of thread-based multitasking as compared to process-based multitasking are:

- Threads share the same address space—that is, global data is accessible to all threads.
- Context switching between threads—that is, switching of execution from one thread to another—is usually less expensive than between processes.
- The cost of coordination between threads is relatively low.

Java supports thread-based multitasking—that is, concurrency—and provides support for multithreaded programming. *Thread-safety* is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

Understanding Concurrency and Parallelism

There is an important distinction between the concept of *parallelism* and *concurrency* when it comes to code execution.

Parallel code execution implies *simultaneous* execution of instructions on different CPU cores. Concurrent execution is a more general idea that does not necessarily imply that the code of different threads is actually executed in parallel. It is possible to create more threads than the CPU cores can execute in parallel, which will cause threads to take turns at sharing time slices on the CPU cores. This means that sometimes the threads are running in parallel and sometimes it only feels like they are because they compete for time slices to execute on a limited number of CPU cores. In other words, concurrent execution gives an illusion of parallel execution.

The JVM is not the only process on the computer that requires CPU time. The operating system and other tasks also share CPU time, making it impossible to predict how much CPU time is allocated and when a thread will get to run, or if any given set of code will be executed in parallel at all. These factors have very important consequences:

1. Execution of code in different threads is unpredictable.
2. A thread that started earlier may or may not complete its work sooner than the thread that started later, even if it has less work to do.
3. Any attempt to control exact execution order will very likely impact performance and may result in all sorts of unwanted side effects, which will be discussed later in this chapter.

The following analogy can be helpful in understanding concurrent code execution:

Imagine that a CPU core is a road and a thread is a car. It would be rather inefficient to build a separate road for every car. Furthermore, one would expect a road to be shared by many cars, which helps to explain why multithreading makes much more efficient use of computer resources. However, even though a given road is shared among many cars, it does not mean that two cars can occupy the same spot on that road at the same time—that is quite obviously an undesirable situation. So cars have to take turns and yield to one another to be able to share the same road, very much like threads have to share a CPU core. It may feel like the cars are using a given road in parallel, but strictly speaking they are actually doing it concurrently. Finally, there is no guarantee that the car that started its journey earlier will get to where it wants to go sooner or later than another car, considering unpredictable circumstances such as being stuck in traffic. This is similar to a concurrent thread that may have to wait for a CPU time slice, if the CPU time has to be shared with many other threads and tasks on the computer. An attempt to control the exact execution order, to ensure that a certain thread gets priority, can be compared to stopping all traffic to yield to an ambulance or a police car—while they get priority, everyone else is stopped, causing overall performance to degrade. It may be tolerable for extraordinary cases, but it would be a traffic disaster if the policy for using the road was to prioritize each and every car on the road. This means that it is best to embrace the stochastic nature of concurrent code execution and try not to control the exact execution order, or at least only do it in exceptional cases.

22.2 Runtime Organization for Thread Execution

Most JVM implementations run as a single process, but allow multiple threads to be created. Runtime organization for thread execution in the JVM is depicted in Figure 22.1. The JVM has designated memory areas, called *runtime data areas*, that are deployed for various purposes during program execution. Figure 22.1 shows data areas that are created specifically for each thread and are private to each thread. The figure also shows data areas that are shared by all threads, called *shared memory*.

Figure 22.1 *Runtime Data Areas*

Each thread has the following data areas, which are private to the thread:

- *JVM stack*: A JVM stack (also known as *execution stack*, *call stack*, or *frame stack*) is created for each thread when the thread starts, and is used for bookkeeping method executions in the thread, as explained in §7.1, p. 365. This is also where all local variables for each active method invocation are stored. Note that each thread takes care of its own exception handling, and thus does not affect other threads.
- *Program counter* (PC): This register is created for each thread when the thread starts, and stores the address of the JVM instruction currently being executed.

The following data areas are shared by all threads:

- *Heap*: This shared memory space is where objects are created, stored, and garbage collected.
- *Method area*: This is created when the JVM starts. It stores the runtime constant pool, field and method information, static variables, and method bytecode for each of the classes and interfaces loaded by the JVM.
- *Runtime constant pool*: In addition to storing the constants defined in each class and interface, this also stores references to all methods and fields. The JVM uses the information in this pool to find the actual address of a method or a field in memory.

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently. Using this powerful paradigm in Java centers on understanding the following aspects of multithreaded programming:

- Creating threads and providing the code that gets executed by a thread (p. 1370)
- Understanding the thread lifecycle (p. 1380)
- Understanding thread issues that occur due to the execution model where concurrent threads are accessing shared memory (p. 1408)

22.3 Creating Threads

A thread in Java is represented by an object of the `java.lang.Thread` class. Every thread in Java is created and controlled by an object of this class. Often the thread (of execution) and its associated `Thread` object are thought of as being synonymous. Implementing the task performed by a thread is achieved in one of two ways:

- Implementing the functional interface `java.lang.Runnable`
- Extending the `java.lang.Thread` class

In both cases, thread creation and management is controlled by the application. Chapter 23, p. 1419, discusses high-level concurrency features that abstract the drudgery of thread creation and management, facilitating the building of massively concurrent applications.

Implementing the `Runnable` Interface

The `Runnable` functional interface has the following specification, comprising a single abstract method declaration:

```
@FunctionalInterface
public interface Runnable {
  void run();
}
```

A thread, which is created based on an object that implements the `Runnable` interface, will execute the code defined in the `public` method `run()`. In other words, the code in the `run()` method defines an independent path of execution and thereby the entry and the exits for the thread. A thread ends when the `run()` method ends, either by normal completion or by throwing an uncaught exception. Note that the method `run()` does not return a value, does not take any parameters, and does not throw any checked exceptions.

The procedure for creating threads based on the `Runnable` interface is as follows:

1. Implement the `Runnable` interface, providing the `run()` method that will be executed by the thread. This can be done by providing a concrete or an anonymous class that implements the `Runnable()` interface. Since the interface is a functional interface, it can also be implemented by a lambda expression, typically for simple tasks.
2. An object of the `Thread` class is created by passing the `Runnable` implementation from step 1 as an argument in the `Thread` constructor call. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in step 2. The `start()` method returns immediately after a thread has been spawned. In other words, the call to the `start()` method is asynchronous.

When the thread, represented by the `Thread` object on which the `start()` method was invoked, gets to run, it executes the `run()` method of the `Runnable` object. This sequence of events is illustrated in Figure 22.2.

In the following code, the functional interface `Runnable` is implemented by a lambda expression that is passed to the `Thread` object. The code will create a thread and start the thread. When the thread gets to run, it will execute the print statement.

Click here to view code image

```
Thread thread = new Thread(
    () -> System.out.println("Harmonious threads create beautiful applications
);
thread.start();
```

Figure 22.2 *Spawning Threads Using a* `Runnable` *Object*

The following is a summary of selected constructors and methods from the `java.lang.Thread` class:

Click here to view code image

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

The argument `threadTarget` is the object whose `run()` method will be executed when the thread is started. The argument `threadName` can be specified to give an explicit name for the thread, rather than an automatically generated one.

```
void start()
```

Spawns a new thread—that is, the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an `IllegalThreadStateException` if the thread is already started or it has already completed execution.

```
void run()
```

The `Thread` class implements the `Runnable` interface by providing an implementation of the `run()` method. This implementation in the `Thread` class does nothing and returns. Subclasses of the `Thread` class should override this method. If the current thread is created using a separate `Runnable` object, the `run()` method of this `Runnable` object is called.

```
static Thread currentThread()
```

Returns a reference to the `Thread` object of the currently executing thread.

Click here to view code image

```
final String getName()
final void setName(String name)
```

The first method returns the name of the thread. The second one sets the thread's name to the specified argument.

Click here to view code image

```
final void setDaemon(boolean flag)
final boolean isDaemon()
```

The first method sets the status of the thread either as a daemon thread or as a normal thread (p. 1377), depending on

> whether the argument is `true` or `false`, respectively. The status should be set before the thread is started. The second method returns `true` if the thread is a daemon thread; otherwise, it returns `false`.

A slightly more elaborate example of creating a thread is presented in Example 22.1. The class `Counter` implements the `Runnable` interface. At (1), the class defines the `run()` method that constitutes the code to be executed in a thread. The `while` loop in the `run()` method executes as long as the current value is less than 5. In each iteration of the `while` loop, the old value and the new incremented value of the counter is printed, as shown at (3). Also, in each iteration, the thread will sleep for 500 milliseconds, as shown at (4). While it is sleeping, other threads may run (p. 1395).

The static method `currentThread()` in the `Thread` class can be used to obtain a reference to the `Thread` object associated with the current thread. We can call the `get-Name()` method on the current thread to obtain its name. An example of its usage, shown at (2), prints the name of the thread executing the `run()` method. Another example of its usage, shown at (5), prints the name of the thread executing the `main()` method.

The `Client` class in Example 22.1 uses the `Counter` class. It creates an object of the class `Counter` at (6). This `Counter` object is passed to two `Thread` objects at (7). The threads are started at (8). In other words, we have two threads that will increment the value in the same `Counter` object.

Both `Thread A` and `Thread B` are *child* threads of the *main* thread. They inherit the normal-thread status from the main thread (p. 1377). The output shows that the main thread finishes executing before the child threads. However, the program will continue running until the child threads have completed their execution of the `run()` method in the `Counter` object.

If a thread has been started or if it has completed, invoking the `start()` method again on the thread results in an `IllegalThreadStateException`, as shown at (9). Note that this does *not* terminate the thread that has already been started. If the thread has completed, a new thread must be created before the `start()` method can be called.

Since thread scheduling is not predictable (p. 1386) and Example 22.1 does not enforce any synchronization between the two threads in accessing the current value in the `Counter` object, the output shown may vary. The output from the two child threads is interspersed. The output also shows that the counter was incremented by 2 when `Thread B` executed for the first time! This is an example of *thread interference*. The challenge in multithreaded programming is to synchronize how shared data is accessed by the threads in the application.

Example 22.1 *Implementing the* `Runnable` *Interface*

Click here to view code image

```
public class Counter implements Runnable {
  private int currentValue = 0;

  @Override
  public void run() {                         // (1) Thread entry point
    String threadName = Thread.currentThread().getName(); // (2)
    while (currentValue < 5) {
      System.out.printf("%s: old:%s new:%s%n",
                        threadName,           // (3) Print thread name,
                        this.currentValue,    //     old value,
                      ++this.currentValue);   //     new incremented value.
      try {
        Thread.sleep(500);                    // (4) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(threadName + " interrupted.");
      }
    }
    System.out.println("Exiting " + threadName);
  }
}
```

Click here to view code image

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();  // (5) main thread
    System.out.println("Method main() runs in thread " + threadName);

    // Create a Counter object:                            // (6)
    Counter counter = new Counter();

    // Create two threads with the same Counter:           // (7)
    Thread threadA = new Thread(counter, "Thread A");
    Thread threadB = new Thread(counter, "Thread B");

    // Start the two threads:                              // (8)
    System.out.println("Starting " + threadA.getName());
    threadA.start();
//  threadA.start();                            // (9) IllegalThreadStateExcep
    System.out.println("Starting " + threadB.getName());
    threadB.start();

    System.out.println("Exiting Thread " + threadName);    // (10)
  }
}
```

Probable output from the program:

Click here to view code image

```
Method main() runs in thread main
Starting Thread A
Starting Thread B
```

```
Exiting Thread main
Thread B: old:0 new:2
Thread A: old:0 new:1
Thread B: old:2 new:3
Thread A: old:3 new:4
Thread B: old:4 new:5
Exiting Thread A
Exiting Thread B
```

Extending the `Thread` Class

A class can also extend the `Thread` class to create a thread. A typical procedure for doing this is as follows (see Figure 22.3):

1. A concrete or an anonymous class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.
2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread, using the `super()` call.
3. The `start()` method inherited from the `Thread` class is invoked on an object of the class to make the thread eligible for running. This method should never be overridden.

Figure 22.3 *Spawning Threads—Extending the* `Thread` *Class*

The simple example below shows an anonymous class that extends the `Thread` class and overrides the `run()` method from the `Thread` superclass. The code will create a thread and start the thread:

Click here to view code image

```
(new Thread() {
   @Override
   public void run() {
     System.out.println("Harmonious threads create beautiful applications.")
   }
 }
).start();
```

In Example 22.2, the `Counter` class from Example 22.1 has been modified to illustrate creating a thread by extending the `Thread` class. The program output shows that the `Client` class creates two threads and exits, but the program continues running until the child threads have completed execution. The two child threads are independent, each having its own counter and executing its own `run()` method. Again note that the main thread finished execution before the child threads, but the JVM does not terminate before the child threads are done.

The `Thread` class implements the `Runnable` interface, which means that this approach is not much different from implementing the `Runnable` interface directly. The main difference is that the roles of the `Runnable` object and the `Thread` object are combined in a single object by extending the `Thread` class.

In the previous two examples, the code to create the `Thread` object, to set the thread name, and to call the `start()` method to initiate the thread execution is in the client code. In Example 22.2, however, setting the name and starting the thread can be placed in the constructor of the `Counter` subclass, as it inherits from the `Thread` class—an exercise the studious reader is encouraged to undertake.

Example 22.2 *Extending the* `Thread` *Class*

Click here to view code image

```
public class Counter extends Thread {
  public int currentValue;
  public Counter() { this.currentValue = 0; }
  public int getValue() { return this.currentValue; }

  @Override
  public void run() {                         // (1) Override from superclass.
    while (this.currentValue < 5) {
      System.out.printf("%s: %s%n",
                   super.getName(),           // (2) Print thread name,
                   this.currentValue++);      //     current value, and increm
      try {
        Thread.sleep(500);                    // (3) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(super.getName() + " interrupted.");
      }
    }
    System.out.println("Exiting " + super.getName());
  }
}
```

Click here to view code image

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();              // (4)
    System.out.println("Method main() runs in thread " + threadName);

    // Create two Counter objects that extend the Thread class:            (5)
    Counter counterA = new Counter();
    Counter counterB = new Counter();

    // Set the names for the two threads:                                  (6)
    counterA.setName("Counter A");
    counterB.setName("Counter B");
    // Mark the threads as daemon threads:                                 (7)
//  counterA.setDaemon(true);
//  counterB.setDaemon(true);

    // Start the two threads:                                          // (8)
    System.out.println("Starting " + counterA.getName());
    counterA.start();
```

```
      System.out.println("Starting " + counterB.getName());
      counterB.start();

      System.out.println("Exiting " + threadName);
    }
  }
```

Probable output from the program:

Click here to view code image

```
Method main() runs in thread main
Starting Counter A
Counter A: 0
Starting Counter B
Exiting main
Counter B: 0
Counter A: 1
Counter B: 1
Counter B: 2
Counter A: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exiting Counter B
Exiting Counter A
```

When creating threads, there are a few reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Extending the `Thread` class means that the subclass cannot extend any other class, whereas a class implementing the `Runnable` interface has this option.
- Extending the `Thread` class also means that the task of the thread—that is, executing the `run()` method—cannot be shared by several threads. This does not mean that the threads cannot access shared resources through their `run()` methods—which is typical in multithreaded applications.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

Types of Threads

The runtime environment distinguishes between *normal threads* (also called *user threads*) and *daemon threads*. As long as a normal thread is alive—meaning that it has not completed executing its task—the JVM does not terminate. A daemon thread is at the mercy of the runtime system: It is stopped if no more normal threads are running, thus terminating the program. Daemon threads exist only to serve normal threads. It is not a good strategy to run any clean-up code in daemon threads, which might not get executed if a daemon thread is stopped.

Uncommenting the statements at (7) in Example 22.2:

Click here to view code image

```
counterA.setDaemon(true);
counterB.setDaemon(true);
```

illustrates the daemon nature of threads. The program execution will now terminate after the main thread has completed, without waiting for the daemon `Counter` threads to finish normally:

Click here to view code image

```
Method main() runs in thread main
Starting Counter A
Starting Counter B
Counter A: 0
Counter B: 0
Exiting main
```

When a standalone application is run, a normal thread is automatically created to execute the `main()` method of the application. This thread is called the *main thread*. If no other normal threads are spawned, the program terminates when the `main()` method finishes executing. All other threads, called *child* threads, are spawned from the main thread, inheriting its normal-thread status. The `main()` method can then finish, but the program will keep running until all normal threads have completed. The status of a spawned thread can be set as either daemon or normal, but this must be done before the thread is *started*. Any attempt to change the status after the thread has been started throws an unchecked `IllegalThreadState-Exception`. A child thread inherits the thread status of its parent thread.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This normal thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have completed after the `main()` method finished executing.

Review Questions

22.1 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class MyClass extends Thread {
  public MyClass(String s) { msg = s; }
  String msg;
  public void run() {
    System.out.println(msg);
  }
  public static void main(String[] args) {
    new MyClass("Hello");
    new MyClass("World");
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors and will print `Hello` and `World`, in that order, every time it is run.

c. The program will compile without errors, and will print a never-ending stream of `Hello` and `World`.

d. The program will compile without errors, and will print `Hello` and `World` when run, but the order is unpredictable.

e. The program will compile without errors, and will simply terminate without any output when run.

22.2 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
class R1 implements Runnable {
  public void run() {
    System.out.print(Thread.currentThread().getName());
  }
}
public class R2 implements Runnable {
  public void run() {
    new Thread(new R1(),"|R1a|").run();
    new Thread(new R1(),"|R1b|").start();
    System.out.print(Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    new Thread(new R2(),"|R2|").start();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors, and will print |`R1a`| twice and |`R2`| once, in some order, every time it is run.

c. The program will compile without errors, and will print|`R1b`| twice and |`R2`| once, in some order, every time it is run.

d. The program will compile without errors, and will print |`R1b`| once and |`R2`| twice, in some order, every time it is run.

e. The program will compile without errors, and will print |`R1a`| once, |`R1b`| once, and |`R2`| once, in some order, every time it is run.

22.3 What will be the result of attempting to compile and run the following program?

Click here to view code image

```
public class Threader extends Thread {
  Threader(String name) {
    super(name);
  }
  public void run() throws IllegalThreadStateException {
    System.out.println(Thread.currentThread().getName());
    throw new IllegalThreadStateException();
  }

  public static void main(String[] args) {
    new Threader("|T1|").start();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without errors, will print |`T1`|, and will terminate normally every time it is run.

c. The program will compile without errors, will print|`T1`|, and will throw an unchecked `IllegalThreadStateException` every time it is run.

d. None of the above

22.4 Thread Lifecycle

Before diving into the lifecycle of a thread, a few important thread-related concepts should be understood.

Objects, Monitors, and Locks

In Java, *each* object has a *monitor* associated with it—including arrays. A thread can *lock* and *unlock* a monitor, but only one thread at a time can *acquire* the lock on a monitor. The thread that acquires the lock is said to *own* the lock and has exclusive access to the object whose monitor was locked. Any other threads trying to acquire the lock are *blocked* and placed in the *entry set* of the monitor until the lock is *released*. At that time, if the entry set is not empty, a blocked thread is allowed to acquire the lock at the discretion of the JVM, based on the thread scheduling policy in effect. The locking mechanism thus implements *mutual exclusion* (also known as *mutex*).

It should be made clear that programs should not make any assumptions about the order in which threads are granted ownership of a lock while waiting in the entry set of the monitor providing the lock. Thread state transitions for lock acquisition are explained later (p. 1388).

The locking mechanism on a monitor is referred to by various names in the literature: *intrinsic lock*, *monitor lock*, *object-level lock*, *monitor*, or just *lock*. Conceptually, the lock is on the object, and therefore we will refer to this locking mechanism provided by a monitor associated with an object as the *object lock* or just *lock*, when it is clear which lock is being referred to.

Classes also have a *class lock* (a.k.a. *class-level lock*) that is analogous to the object lock. Such a lock is actually a lock on the `java.lang.Class` object that represents the class at runtime in the JVM. Given a class `A`, the reference `A.class` denotes this unique `Class` object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The following `static` method of the `Thread` class can be used to determine whether the current thread holds a lock on a specific object:

> Click here to view code image
>
> ```
> static boolean holdsLock(Object obj)
> ```

Thread States

Understanding the lifecycle of a thread can be beneficial when programming with threads. Threads can exist in different states. Just because the `start()` method has been called on the thread, it does not mean that the thread has access to the CPU and can start executing straightaway. Several factors determine how it will proceed.

Figure 22.4 shows the states and the transitions in the lifecycle of a thread. The figure shows the main transitions that occur during the lifecycle of a thread—in particular, transitions pertaining to *blocked I/O* (p. 1405) and *thread interruption* (p. 1393) have been omitted.

The thread states are represented by enum constants and summarized in Table 22.1. The `Thread` class provides the `getState()` method to determine the state of the current thread. The method returns a constant of type `Thread.State` (i.e., the type `State` is a static inner `enum` type declared in the `Thread` class).

Figure 22.4 *Thread States*

Table 22.1 *Thread States Defined by the* `Thread.State` *Enum Type*

Constant in the `Thread.State`	**Description of the state**
`NEW`	A thread in this state has been created, but it has not yet started. A thread is started by calling its `start()` method (p. 1370).
`RUNNABLE`	The `RUNNABLE` states can be characterized by two *substates* which are not observable: • *READY substate*: A thread starts life in the *READY* substate (p. 1386), and also when it transitions from a *non-runnable state* to the `RUNNABLE` state. • *RUNNING substate*: If a thread is in the *RUNNING* substate, it means that the thread is currently executing (p. 1386), and is deemed the *running thread*. From the *RUNNING* substate, a thread can transition either to the *READY* substate, to a non-runnable state, or to the `TERMINATED` state.
`BLOCKED`	A thread is blocked while waiting to acquire a lock (p. 1380, p. 1388). A thread is blocked while waiting for I/O (p. 1405).
`WAITING`	A thread is in this state indefinitely for the following reasons, until the action it is waiting for occurs: • A thread is waiting indefinitely for join completion: The thread awaits completion of another thread (p. 1402). • A thread is waiting indefinitely for notification: The thread awaits notification from another thread (p. 1396).
`TIMED_WAITING`	A thread is in this state for at least a *specified amount of time* for the following reasons, unless the action it is waiting for occurs before timeout: • Timed waiting for join completion: The thread awaits

completion of another thread (p. 1402).

- Timed waiting for notification: The thread awaits notification from another thread (p. 1396).
- Timed waiting to wake up from sleep (p. 1395).

`TERMINATED` The `run()` method completed execution or terminated. Once in this state, the thread can never run again (p. 1405).

The rest of this section will expound on the thread states and the transitions between them during the lifecycle of a thread. However, the following remarks on the thread lifecycle in Figure 22.4 should be noted.

- The `RUNNABLE` state is a *compound state* having two *non-observable substates*: *READY* and *RUNNING*. Being *non-observable* means it is not possible to distinguish which substate the thread is in once it is in the `RUNNABLE` state.
- The following states are characterized as *non-runnable* states: `BLOCKED`, `WAITING`, and `TIMED_WAITING`. A *running thread*—one in the *RUNNING* substate—can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a specific transition occurs. A thread does not go directly to the *RUNNING* substate from a non-runnable state, but transits first to the *READY* substate.

Selected methods from the `Thread` class are presented below. Examples of their usage are presented in subsequent sections.

```
final boolean isAlive()
```

This method can be used to find out if a thread is alive or dead. A thread is *alive* if it has been started but not yet terminated—that is, it is not in the `TERMINATED` state.

```
Thread.State getState()
```

Returns the state of this thread (Table 22.1). It should be used for monitoring the state and not for synchronizing control.

Click here to view code image

```
final int getPriority()
final void setPriority(int newPriority)
```

The first method returns the priority of a thread (p. 1385). The second method changes its priority. The priority set will be the minimum of the specified `newPriority` and the maximum priority permitted for this thread. There is no guarantee that a thread with a higher priority will be chosen to run.

```
static void yield()
```

Causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute. It is up to the JVM to decide if and when this transition will take place. The transition is from the *RUNNING* substate to the *READY* substate in the `RUNNABLE` state (Figure 22.4).

Click here to view code image

```
static void sleep(long millisec) throws InterruptedException
static void sleep(long millis, int nanos) throws InterruptedException
```

The current thread sleeps for at least the specified time before it becomes eligible for running again. The transition is from the *RUNNING* substate of the `RUNNABLE` state to the `TIMED_WAITING` state (Figure 22.4).

Click here to view code image

```
final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException
```

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively. In Figure 22.4, the transition is from the *RUNNING* substate of the `RUNNABLE` state to the `WAITING` state for the first method and to the `TIMED_WAITING` state for the second method.

Click here to view code image

```
static Map<Thread, StackTraceElement[]> getAllStackTraces()
```

Returns a map of stack traces for all alive threads. Keep in mind that each thread has its own JVM stack for method execution. The map can be used, for example, to extract a set of all alive threads.

Example 22.3 illustrates transitions between thread states. A thread at (1) sleeps a little at (2) and then does some computation in a loop at (3), after which the thread terminates. The `main()` method monitors the thread in a loop at (4), printing the thread state returned by the `getState()` method. The output shows that the thread starts off in the `NEW` state and goes through the `RUNNABLE` state when the `run()` method starts to execute, and then transits to the `TIMED_WAITING` state to sleep. On waking up, it computes the loop in the `RUNNABLE` state, and transits to the `TERMINATED` state when the `run()` method completes execution.

Example 22.3 *Thread States*

Click here to view code image

```
public class ThreadStates {

  private static Thread t1 = new Thread("T1") {     // (1)
    @Override public void run() {
      try {
        sleep(1);                                   // (2)
        for (int i = 10000; i > 0; i--) {;}         // (3)
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  };

  public static void main(String[] args) {
    System.out.println(t1.getState());               // (4)
    t1.start();
    while (true) {                                   // (5)
      Thread.State state = t1.getState();
      System.out.println(state);
      if (state == Thread.State.TERMINATED) {
        break;
      }
    }
  }
}
```

Probable output from the program:

```
NEW
RUNNABLE
TIMED_WAITING
...
TIMED_WAITING
RUNNABLE
...
RUNNABLE
TERMINATED
```

Thread Priorities

Threads are assigned priorities that the thread scheduler *can* use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the *READY* substate. This is not necessarily the thread that has been in the *READY* substate the longest time. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Selective priorities are defined by `enum` constants in the `Thread` class, shown in Table 22.2.

Table 22.2 *Selective Priorities Defined by the* `Thread` *Class*

Field	**Type** `java.lang.Thread`	**Value**	**Description**
`MIN_PRIORITY`		`0`	Lowest priority
`NORM_PRIORITY`		`5`	Default priority
`MAX_PRIORITY`		`10`	Highest priority

A thread inherits the priority of its parent thread. The priority of a thread can be set using the `setPriority()` method and read using the `getPriority()` method, both of which are defined in the `Thread` class. The following code sets the priority of the thread `myThread` to the minimum of two values: maximum priority and current priority incremented to the next level. It also shows the default text representation of a thread, `[`*`thread_name`*`,` *`priority`*`,` *`parent_thread_name`*`]`.

Click here to view code image

```
Thread myThread = new Thread(() ->
    System.out.println(Thread.currentThread() + ": Don't mess with my priority
);
System.out.println(myThread);
myThread.setPriority(Math.min(Thread.MAX_PRIORITY, myThread.getPriority() + 1)
myThread.start();
```

Output from the code:

Click here to view code image

```
Thread[Thread-0,5,main]
Thread[Thread-0,6,main]: Don't mess with my priority!
```

The `setPriority()` method is an *advisory* method, meaning that it provides a hint from the program to the JVM, which the JVM is in no way obliged to honor. A thread with higher priority is not guaranteed to complete its work faster than a thread with lower priority. The method can be used to fine-tune the *performance* of the program, but should not be relied upon for the *correctness* of the program.

Thread Scheduler

Schedulers in JVM implementations usually employ one of the following two strategies, which come into play in the `RUNNABLE` state:

- Preemptive scheduling

  If a thread with a higher priority than the current running thread moves to the *READY* substate, the current running thread can be *preempted* (moved from the *RUNNING* substate to the *READY* substate) to let the higher-priority thread execute.

- Time-sliced or round-robin scheduling

  A running thread is allowed to execute for a fixed length of time in the *RUNNING* substate, after which it moves to the *READY* substate to await its turn to run again.

It should be emphasized that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Starting a Thread

After a thread has been created, the thread is in the `NEW` state (Figure 22.5). Only after its `start()` method has been called, can the thread do useful work. This method is asynchronous—that is, it returns immediately. The newly created thread transits to the *READY* substate, waiting its turn to execute on the CPU at the discretion of the thread scheduler. The `getState()` method on this thread will return the value `Thread.State.RUNNABLE`. For a thread that is created but not started, the `getState()` method will return the value `Thread.State.NEW`. Details of creating and starting a thread were covered in an earlier section (p. 1370).

Figure 22.5 *Starting a Thread*

Running and Yielding

Once in the *READY* substate, the thread is eligible for running—that is, it waits for its turn to get CPU time. The thread scheduler decides which thread runs and for how long. Figure 22.6 illustrates the transitions between the *READY* and *RUNNING* substates. The thread transits from the *READY* substate to the *RUNNING* substate when it is its turn to run. The `getState()` method on this thread will return the value `Thread.State.RUNNABLE`.

A call to the static method `yield()`, defined in the `Thread` class, may cause the current thread in the *RUNNING* substate to transit to the *READY* substate. If this happens, the thread is then at the mercy of the thread scheduler as to when it will run again. It is possible that if there are no threads in the *READY* substate, this thread can continue executing. If there are other threads in the *READY* substate, their priorities can influence which thread gets to execute.

As with the `setPriority()` method, the `yield()` method is also an advisory method, and therefore comes with no guarantees that the JVM will honor the call. A call to the `yield()` method does *not* affect any locks that the thread might hold.

Figure 22.6 *Running and Yielding*

By calling the static method `yield()`, the running thread gives other threads in the *READY* substate a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of cancelling it by clicking a CANCEL button. If the computation thread hogs the CPU and the user clicks the CANCEL button, chances are that it might take awhile before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following `run()` method:

Click here to view code image

```
public void run() {
  try {
    while (!done()) {
      doLittleBitMore();
      Thread.yield();                // Current thread yields.
    }
  } catch (InterruptedException ie) { // Clean up if the thread is interrupted
    doCleaningUp();
  }
}
```

Executing Synchronized Code

Threads share the same memory space—that is, they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently among several users without proper discipline will jeopardize the integrity of the account data. Java provides the concept of *synchronization* in order to control access to shared resources. It is one of the mechanisms that Java provides to write *thread-safe code* (§23.4, p. 1451).

The keyword `synchronized` and the intrinsic lock mechanism (p. 1380) form the basis for implementing synchronized execution of code. We refer to this as *intrinsic locking* (also referred to as *built-in synchronization*). There are two ways in which execution of code can be synchronized: by declaring `synchronized` *methods* or by using `synchronized` *statements*. The code that is synchronized to execute by one thread at a time is often referred to as a *critical section*.

Acquiring the Object Lock

In order to execute `synchronized` code, a thread must first acquire the lock of the relevant object, as illustrated in Figure 22.7. If this lock is not available, the thread transits from the *RUNNING* substate to the `BLOCKED` state. The thread is put in the *entry set* of the object. Threads in the `BLOCKED` state are grouped according to the object whose lock they are waiting for. The `getState()` method on a blocked thread in this state will return the value `Thread.State.BLOCKED`.

If there are several threads waiting for the same object lock, one is chosen at the discretion of the thread scheduler. It then transits to the *READY* substate where it takes its turn to run again.

Example 22.4 illustrates acquiring the lock in order to synchronize on a shared resource.

Figure 22.7 *Blocked for Lock Acquisition*

Synchronized Methods

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword `synchronized`. A thread wishing to execute a `synchronized` method must first acquire the object lock before it can execute the method on the object. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits in the entry set of the object lock, as explained earlier. No particular action on the part of the program is necessary to acquire the lock. A thread releases the lock on method return, regardless of whether the `synchronized` method completed execution normally or threw an uncaught exception. In both cases, if there are any blocked threads in the entry set, one is chosen to acquire the lock. In case of an uncaught exception, the exception is propagated through the JVM stack of the thread that released the lock.

Synchronized methods are useful in situations where methods can manipulate the state of a shared object in ways that can corrupt the state if executed concurrently.

Example 22.4 is a reworking of the counter implementation from Example 22.1. The `CounterX` class defines the `synchronized` method `increment()` at (1a) that prints the old value and the new incremented value in the counter.

The `main()` method in the `SynchronizedMethodDemo` class creates a `CounterX` object at (2) and a `Runnable` at (3) that calls the `increment()` method five times on this `CounterX` object. Two threads are created and started at (4) and (5) that will each call the synchronized `increment()` method on the *same* `CounterX` object five times—in contrast to Example 22.1, where each thread had its own counter.

From the output shown in Example 22.4, we can see that the main thread exits right after creating and starting the threads. The output shows that the two threads run mutually exclusive of each other. `Thread-0` increments the counter to 5 and `Thread-1` increments it further to 10.

It is instructive to run Example 22.4 with (1a) commented out and (1b) uncommented. We see from the output that the execution of the threads is interleaved, and some results look dubious (`Thread-1: old:0 new:2`). Non-`synchronized` incrementing of the value in the counter between the two threads is a disaster waiting to happen. This is an example of what is called a *race condition* (also known as *thread interference*). It occurs when two or more threads simultaneously update a shared value and, due to the scheduling of the threads, can leave the value in an undefined or inconsistent state.

Running the program in Example 22.4 with the `synchronized` version of the `increment()` at (1a) avoids any race conditions. The lock is only released when the `synchronized` method exits, guaranteeing a mutually exclusive increment of the counter by each thread.

Example 22.4 *Mutual Exclusion*

Click here to view code image

```
class CounterX {
  private int counter = 0;

  public synchronized void increment() {                            // (1a)
//public void increment() {                                         // (1b)
    System.out.println(Thread.currentThread().getName()
                     + ": old:" + counter + " new:" + ++counter);
  }
}
```

```
public class SynchronizedMethodDemo {
  public static void main(String[] args) {
    CounterX counter = new CounterX();                              // (2)
    Runnable r = () -> {                                            // (3)
      for (int i = 0; i < 5; i++) {
        counter.increment();
      }

      System.out.println("Exiting " + Thread.currentThread().getName());
    };
    new Thread(r).start();                                          // (4)
    new Thread(r).start();                                          // (5)
    System.out.println("Exiting thread " + Thread.currentThread().getName());
  }
}
```

Probable output from the program when run with (1a):

```
Exiting thread main
Thread-0: old:0 new:1
Thread-0: old:1 new:2
Thread-0: old:2 new:3
Thread-0: old:3 new:4
Thread-0: old:4 new:5
Exiting Thread-0
Thread-1: old:5 new:6
Thread-1: old:6 new:7
Thread-1: old:7 new:8
Thread-1: old:8 new:9
Thread-1: old:9 new:10
Exiting Thread-1
```

Probable output from the program when run with (1b):

```
Thread-1: old:0 new:2
Thread-1: old:2 new:3
Exiting thread main
Thread-0: old:0 new:1
Thread-1: old:3 new:4
Thread-0: old:4 new:5
Thread-0: old:6 new:7
Thread-1: old:5 new:6
Thread-1: old:8 new:9
Exiting Thread-1
Thread-0: old:7 new:8
Thread-0: old:9 new:10
Exiting Thread-0
```

Reentrant Synchronization

While a thread is inside a `synchronized` method of an object, all other threads that wish to execute this `synchronized` method or any other `synchronized` method of the object will have to wait in the entry set. This restriction does not apply to the thread that already has the lock and is executing a `synchronized` method of the object. Such a thread can invoke, directly or indirectly, other `synchronized` methods of the object without being blocked—that is, once a thread has a lock on an object, it can acquire the lock on the same object several times, called *reentrant synchronization*. The non-`synchronized` methods of the object can always be called at any time by any thread.

Examples of reentrant synchronization can be found in §23.4, p. 1460.

Synchronization on Class Lock

Static methods synchronize on the *class lock*. Acquiring and releasing a class lock by a thread in order to execute a `static synchronized` method is analogous to that of an object lock for a `synchronized` instance method. A thread acquires the class lock before it can proceed with the execution of any static `synchronized` method in the class, blocking other threads wishing to execute any `static synchronized` methods in the same class. The blocked threads wait in the entry set of the class lock. This does not apply to static, non-`synchronized` methods, which can be invoked by any thread. A thread acquiring the lock of a class to execute a `static synchronized` method has no effect on any thread acquiring the lock on any object of the class to execute a `synchronized` instance method. In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

A subclass decides whether the new definition of an inherited `synchronized` method will remain `synchronized` in the subclass.

Synchronized Statements

Whereas execution of `synchronized` methods of an object is synchronized on the lock of the object, the `synchronized` statement allows execution of arbitrary code to be synchronized on the lock of an *arbitrary object*. The general form of the `synchronized` statement (also called *synchronized block*) is as follows:

Click here to view code image

```
synchronized (object_reference_expression) { code_block }
```

The *object reference expression* must evaluate to a non-null reference value; otherwise, a `NullPointerException` is thrown. The *code block* is usually related to the object whose lock is acquired. This is analogous to a `synchronized` method, where the execution of the method is synchronized on the lock of the current object. The following code is equivalent to the `synchronized increment()` method in the `CounterX` class in Example 22.4:

Click here to view code image

```
public void increment() {
  synchronized(this) {                               // Synchronized statemen
    System.out.println(Thread.currentThread().getName()
        + ": current: " + counter + " new:" + ++counter);
  }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is released. This happens when the execution of the code block completes normally or an uncaught exception is thrown. In contrast to `synchronized` methods, this mechanism allows fine-grained synchronization of code on arbitrary objects.

The object reference expression in the `synchronized` statement is mandatory. A class can choose to synchronize the execution of a part of a method by using the `this` reference and putting the relevant code of the method in the `synchronized` statement.

The curly brackets of the block cannot be omitted, even if the code block has just one statement.

Click here to view code image

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
    synchronized (account) {       // (1) synchronized statement
      account.update();            // (2)
    }
  }
}
```

In the example above, the code at (2) in the `synchronized` statement at (1) is synchronized on the `BankAccount` object. If several threads were to concurrently execute the method `updateTransaction()` on an object of `SmartClient`, the statement at (2) would be executed by one thread at a time only after synchronizing on the `BankAccount` object associated with this particular instance of `SmartClient`.

Inner classes can access data in their enclosing context (§9.1, p. 491). An inner object might need to synchronize on its associated outer object in order to ensure integrity of data in the latter. This is illustrated in the following code where the `synchronized` statement at (5) uses the special form of the `this` reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method `incr()` in an inner object can only access the `private double` field `count` at (2) in the `synchronized` statement at (5) by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be released before it can proceed with the execution of the `synchronized` statement at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

Click here to view code image

```
class Outer {                            // (1) Top-level Class
  private double count;                  // (2)
  protected class Inner {                // (3) Non-static member Class
    public void incr() {                 // (4)
      synchronized(Outer.this) {         // (5) Synchronized statement on Outer obje
        ++count;                         // (6)
      }
    }
  }
}
```

A `synchronized` statement can also be specified on a class lock:

Click here to view code image

```
synchronized (<class_name>.class) { <code_block>}
```

The statement synchronizes on the lock of the object denoted by the reference *`class_name`*`.class`. This object (of type `Class`) represents the class at runtime. A `static synchronized` method `classAction()` in class `A` is equivalent to the following declaration:

Click here to view code image

```
static void classAction() {
  synchronized (A.class) {          // Synchronized statement on class A
    // ...
  }
}
```

In summary, a thread can acquire a lock by carrying out the following actions:

- By executing a `synchronized` instance method of an object of a class.
- By executing a `static synchronized` method of a class (in which case, the object is the `Class` object representing the class in the JVM).
- By executing the body of a `synchronized` statement that synchronizes either on an object of a class, or on the `Class` object representing a class in the JVM.
- Intrinsic locking does not provide any guarantees as to which thread among the waiting threads will acquire the intrinsic lock.
- Intrinsic lock acquisition is rigid: Either the thread acquires the intrinsic lock immediately if it is available, or the thread has to wait its turn among the waiting threads. There is no other lock acquisition policy—for example, timed waiting.
- An intrinsic lock is acquired and released in the same synchronized code—for example, in a synchronized method. It cannot be acquired or released in separate methods.

Interrupt Handling

The purpose of interrupts is to allow threads to inform each other when the task they are running might need attention. For example, a thread might be hoarding crucial resources or might have become unresponsive. Whatever corrective action is required is then at the discretion of the thread that is interrupted. Simply catching and ignoring an `InterruptedException` is not recommended.

The following selected methods from the `Thread` class can be used for handling interrupts in threads.

> ```
> void interrupt()
> ```
>
> Interrupts the thread on which it is invoked.
>
> If the thread is in a non-runnable state, either blocked or waiting, the *interrupt status* of the thread is *cleared* and the thread will receive an `InterruptedException` when it gets to run.

> For a thread that is in the `RUNNABLE` state, the *interrupt status* of the thread will be *set*.
>
> ```
> boolean isInterrupted()
> ```
>
> Checks whether this thread has been interrupted. The *interrupt status* of the thread is *not* affected.
>
> ```
> static boolean interrupted()
> ```
>
> Checks whether the current thread has been interrupted. The *interrupt status* of the thread is cleared.

A thread can be interrupted by invoking the method `interrupt()`. A thread can also invoke this method on itself. How the interrupt manifests in the interrupted thread depends on the state the thread is in when the `interrupt()` method is invoked. Example 22.5 illustrates the two main scenarios that show how a thread can discover if it has been interrupted—and which a thread should be prepared to handle.

In Example 22.5, the `main()` method creates and starts a thread at (1) that executes the task defined at (3). The main thread uses an infinite loop at (2) in which it tries to send an interrupt to the `worker` thread by calling the `interrupt()` method. Sending an interrupt is dependent on the condition in the `if` statement being a random even number. If it cannot send an interrupt, the main thread sleeps for awhile before continuing to execute the infinite loop. The main thread exits the infinite loop (and terminates) when it has sent an interrupt to the `worker` thread.

The task executed by the `worker` thread is defined by the lambda expression at (3). The task executes an infinite `while` loop at (4) that illustrates how a thread can discover it has received an interrupt:

- An `if` statement at (5), which is used to determine if the thread has been interrupted. This corresponds to the first scenario in the output.

  If any interrupt is sent to the current thread by calling its `interrupt()` method while it is running, the interrupt status of the thread will be set. Calling the `isInterrupted()` method on the current thread in the `if` statement will then return `true`.
- A `try-catch` block at (6) in which the thread sleeps, and

catches any `Interrupted-Exception` that is thrown in response to receiving an interrupt while it was asleep. This corresponds to the second scenario in the output.

If the `interrupt()` method was called on the thread while it was sleeping, it will result in the interrupt status being cleared and an `InterruptedException` being thrown in the `try` block when the thread runs again. Appropriate action can be taken to mitigate the interrupt in the `catch` block at (7) in which this exception is caught.

Example 22.5 *Interrupt Handling*

Click here to view code image

```
public class InterruptHandling {
  public static void main(String[] args) throws InterruptedException {
    Thread worker = new Thread(task, "worker");                       // (1)
    worker.start();
    while (true) {                                                    // (2)
      if ((int)(Math.random()*100) % 2 == 0) {
        worker.interrupt();
        break;
      }
      Thread.sleep(2);
    }
  }

  private static Runnable task = () -> {                              // (3)
    Thread ct = Thread.currentThread();
    String threadName = ct.getName();
    while (true) {                                                    // (4)
      System.out.println(threadName + " performing task");

      if (ct.isInterrupted()) {                                       // (5)
        System.out.println(threadName + ": interrupted flag is "
                         + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }

      try {                                                            // (6)
        Thread.sleep(2);
      } catch (InterruptedException e) {                               // (7)
        System.out.println(threadName + " caught " + e);
        System.out.println(threadName + ": interrupted flag is "
                         + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }
    }
  };
}
```

Probable output from the program—first scenario:

Click here to view code image

```
worker performing task
worker: interrupted flag is true
worker terminating
```

Probable output from the program—second scenario:

Click here to view code image

```
worker performing task
worker performing task
worker caught java.lang.InterruptedException: sleep interrupted
worker: interrupted flag is false
worker terminating
```

Sleeping and Waking Up

Transitions by a thread that is sleeping and waking up are illustrated in Figure 22.8.

A call to the static method `sleep()` in the `Thread` class will cause the *currently running* thread to temporarily pause its execution and transit to the `TIMED_WAITING` state. The `getState()` method on this thread will return the value `Thread.State.TIMED_WAITING`.

The thread is *guaranteed* to sleep for *at least* the time specified in its argument, before transitioning to the *READY* substate where it takes its turn to run again. It might sleep a little longer than the specified time, but it will not sleep indefinitely.

Figure 22.8 *Sleeping and Waking Up*

If a thread is interrupted while sleeping, it will throw a checked `Interrupted-Exception` when it awakes and gets to execute. The idiom is to call the `sleep()` method in a `try-catch` block that handles the `InterruptedException`. The `sleep()` method does *not* release any locks that the thread might have.

There are two overloaded versions of the `sleep()` method in the `Thread` class, allowing time to be specified in milliseconds, and additionally in nanoseconds.

Usage of the `sleep()` method is illustrated in Examples 22.1, 22.2, and 22.4.

Thread Coordination

Before proceeding further with thread coordination using the `wait()` and `notify()/ notifyAll()` methods, it must be emphasized that these methods provide *low-level* thread coordination, which is error-prone and difficult to get right, and should be avoided in favor of the *high-level* concurrency support provided by the `java.util.concurrent` package (Chapter 23, p. 1419).

Waiting and notifying provide a means of coordination between threads that *synchronize on the same object*—also called *data access synchronization*. The threads execute `wait()` and `notify()` (or `notifyAll()`) methods *on the shared object* for this purpose. These `final` methods are defined in the `Object` class, and therefore are inherited by all objects. These methods can only be executed on an object whose lock the thread holds (i.e., in `synchronized` code); otherwise, the call will result in an `IllegalMonitorStateException`.

> Click here to view code image
>
> ```
> final void wait(long timeout) throws InterruptedException
> final void wait(long timeout, int nanos) throws InterruptedException
> final void wait() throws InterruptedException
> ```
>
> A thread invokes the `wait()` method on the object whose lock it holds. The thread is added to the *wait set* of the current object.
>
> ```
> final void notify()
> final void notifyAll()
> ```
>
> A thread invokes a notification method on the current object whose lock it holds to notify thread(s) that are in the wait set of the object.

Waiting

Coordination between threads is facilitated by waiting and notifying, as illustrated by Figures 22.9 and 22.10. A thread usually calls the `wait()` method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the *RUNNING* substate and transits to the `WAITING` or `TIMED_WAITING` state, depending on whether a timeout was specified in the call. The thread releases ownership of the object lock.

Figure 22.9 *Timed Waiting and Notifying*

Transitioning to a waiting state and releasing the object lock are completed as one *atomic* (non-interruptible) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object, after acquiring its lock.

Note that the waiting thread releases only the lock of the object on which the `wait()` method was invoked. It does not release any other object locks that it might hold, which will remain locked while the thread is waiting.

Each object has a *wait set* containing threads waiting for notification. Threads in a waiting state are grouped according to the object whose `wait()` method they invoked.

Figure 22.10 shows a thread t_1 that first acquires a lock on the shared object, and afterward invokes the `wait()` method on the shared object. This releases the object lock, the thread t_1 is added to the wait set of the object, and the thread waits to be notified in the `WAITING` state. While the thread t_1 is waiting, another thread t_2 can acquire the lock on the shared object for its own purpose.

Depending on whether a timeout was specified or not in the call to the `wait()` method, a call to the `getState()` method on thread t_1 while it is in a waiting state will return the value `Thread.State.TIMED_WAITING` or `Thread.State.WAITING`, respectively.

A thread in a waiting state can be awakened by the occurrence of any one of these events:

- Another thread invokes the `notify()` method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
- The waiting thread times out if it was a timed wait.
- Another thread interrupts the waiting thread.
- A spurious wakeup occurs.

Figure 22.10 *Thread Coordination*

Notified

Invoking the `notify()` method on an object wakes up a single thread that is waiting for the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM. On being *notified*, a waiting thread first transits to the `BLOCKED` state to acquire the lock on the object, and not directly to the *READY* substate of the `RUNNABLE` state. The thread is also removed from the wait set of the object. Note that the object lock is not released when the notifying thread invokes the `notify()` method. The notifying thread releases the lock at its own discretion, and the awakened thread will not be able to run until the notifying thread releases the object lock.

When the notified thread obtains the object lock, it is enabled for execution, waiting in the *READY* substate for its turn to execute again. Finally, when it does execute, the call to the `wait()` method returns and the thread can continue with its execution.

From Figure 22.10 we see that thread t_2 does not release the object lock when it invokes the `notify()` method. Thread t_1 is forced to wait in the `BLOCKED` state for lock acquisition. It is shown no privileges, and must compete with any other threads blocked for lock acquisition.

A call to the `notify()` method has no effect if there are no threads in the wait set of the object.

In contrast to the `notify()` method, the `notifyAll()` method wakes up *all* threads in the wait set of the shared object. They will all transit to the `BLOCKED` state, and contend for the object lock as explained earlier.

It should be stressed that a program should not make any assumptions about the order in which threads awaken in response to the `notify()` or `notifyAll()` method, or transit to the `BLOCKED` state for lock acquisition.

Timed-out

The `wait()` call can specify the time the thread should wait before being timed out, if it was not awakened by being notified or interrupted. The timed-out thread transits to the `BLOCKED` state and competes to acquire the lock as explained above. Note that the awakened thread has no way of knowing whether it was timed out or awakened by one of the notification methods.

If no timeout is specified in the call to the `wait()` method, the thread waits indefinitely in the `WAITING` state until it is notified, interrupted, or awakened spuriously.

Interrupted

This means that another thread invoked the `interrupt()` method on the waiting thread. The awakened thread is enabled as previously explained, but the return from the `wait()` call will result in an `InterruptedException` if and when the awakened thread finally gets a chance to run. The code invoking the `wait()` method must be prepared to handle this checked exception (p. 1393).

Spurious Wakeup

In very rare cases, a thread in a waiting state is awakened by what is called a *spurious wakeup*, which is not due to familiar events like being notified, timed out, or interrupted, but due to deep-level system signals warranting a wakeup call to the waiting threads. The condition on which the thread was waiting might not have been fulfilled when the thread wakes up from a spurious wakeup, and should therefore be rechecked when the thread gets to run. This is done by testing the condition in a loop that contains the `wait()` call, as the thread can go right back to waiting if the condition is not satisfied.

Click here to view code image

```
while (conditionNotSatisfied()) {
  ...
  wait();
  ...
}
// Proceed ...
```

Using Wait and Notify

The idiom is to call the `wait()` method in a `try-catch` block inside a loop, guarded by a condition that the thread expects to be set by notification.

In Example 22.6, the `main()` method of the class `WaitNotifyScenario` creates and starts four daemon threads that are manipulating the same `MessageDisplay` object. Two of them are continuously trying to set a message in the message display, while the other two are continuously trying to print whatever message is in the display.

Since the threads manipulate the same `MessageDisplay` object, and the `setMessage()` and `displayMessage()` methods in the class `MessageDisplay` are `synchronized`, it means that the threads synchronize on the same `MessageDisplay` object. In other words, the mutual exclusion of these operations is guaranteed on the same `MessageDisplay` object.

Example 22.6 *Waiting and Notifying*

Click here to view code image

```
class MessageDisplay {
  private String message;
  public synchronized void displayMessage() {
    String threadName = Thread.currentThread().getName();
    while (this.message == null) {                           // No message?
      try {
        wait();                                              // (1)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println(threadName + ": " + this.message);    // Display messag
    this.message = null;                                     // Remove message
    notifyAll();                                             // (2)
  }

  public synchronized void setMessage(String message) {
    String threadName = Thread.currentThread().getName();
    while (this.message != null) {                           // Message presen
      try {
        wait();                                              // (3)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.message = message;                                  // Set new messag
    System.out.println(threadName + ": Message set is " + this.message);
    notifyAll();                                             // (4)
  }
}
```

```
public class WaitNotifyScenario {
  public static void main(String[] args) throws InterruptedException {
    MessageDisplay md = new MessageDisplay();
    Thread t1 = new Thread(() -> { while (true) md.setMessage("Hi!"); }, "t1")
    t1.setDaemon(true);
    t1.start();
    Thread t2 = new Thread(() -> { while (true) md.setMessage("Howdy!"); }, "t
    t2.setDaemon(true);
    t2.start();
    Thread t3 = new Thread(() -> { while (true) md.displayMessage(); }, "t3");
    t3.setDaemon(true);
    t3.start();
    Thread t4 = new Thread(() -> { while (true) md.displayMessage(); }, "t4");
    t4.setDaemon(true);
    t4.start();
    Thread.sleep(5);
    System.out.println("Exit from main.");
  }
}
```

Probable output from the program (*edited to fit on the page*):

```
t0: t1: Message set is Hi!
t3: Hi!
...
t2: Message set is Howdy!
t3: Howdy!
t1: Message set is Hi!
t3: Hi!
t2: Message set is Howdy!
t4: Howdy!
...
Exit from main.
t1: Message set is Hi!
t3: Hi!
t2: Message set is Howdy!
t4: Howdy!
t1: Message set is Hi!
```

Example 22.6 illustrates how threads waiting as a result of calling the `wait()` method on an object are notified by another thread calling the `notifyAll()` method on the same object, in order for a waiting thread to start running again.

One usage of the `wait()` call is shown in Example 22.6 at (1) in the `synchronized displayMessage()` method. When a thread executing this method on the `Message-Display` object finds that there is no message (`this.message == null`), it invokes the `wait()` method in order to wait for a thread to set a message.

Another use of the `wait()` call is shown at (3) in the `synchronized setMessage()` method. When a thread executing this method on the `MessageDisplay` object finds that there is already a message (`this.message != null`), it invokes the `wait()` method to wait for a thread to display and remove the message.

When a thread executing the `synchronized` method `setMessage()` on the `Message-Display` object successfully sets a message, it calls the `notifyAll()` method at (4). The wait set of the `MessageDisplay` object may contain any waiting threads that have earlier called the `wait()` method at either (1) or (3) on this `MessageDisplay` object. A single thread from the wait set is enabled for running. If this thread was executing a `displayMessage()` operation, it now has a chance of being successful because a message is available for display at the moment. If this thread was executing a `setMessage()` operation, it can try again to see if the previous message has been removed so that a new message can be set.

When a thread executing the `synchronized` method `displayMessage()` on the `Message-Display` object successfully prints and removes the message, it calls the `notifyAll()` method at (2). Again assuming that the wait set of the `MessageDisplay` object is not empty, one thread from the set is arbitrarily chosen and enabled. If the notified thread was executing a `setMessage()` operation, it now has a chance of succeeding because there is no message in the `MessageDisplay` object at the moment.

Note that the waiting condition at (1) for the `displayMessage()` operation is executed in a loop. A waiting thread that has been notified is not guaranteed to run right away. Before it gets to run, another thread may synchronize on the `MessageDisplay` object and remove the message. If the notified thread was waiting to display the message, it would now incorrectly try to display a nonexisting message because the condition was not tested after notification. The loop ensures that the condition is always tested after notification, sending the thread back to the waiting state if the condition is not met. To avert the analogous danger of setting a message if one already exits in the `MessageDisplay` object, the waiting condition at (3) for the `setMessage()` operation is also executed in a loop.

The output from Example 22.6 shows the behavior of the threads that set and print the message in the shared `MessageDisplay` object, respectively. The two operations of setting the message and displaying/removing the message are performed in lockstep.

The four threads created are daemon threads. They will be terminated if they have not completed when the main thread dies, thereby stopping the execution of the program.

Joining

A thread can invoke the overloaded method `join()` (from the `Thread` class) on another thread in order to wait for the other thread to complete its execution before continuing—that is, the first thread waits for the second thread to *join it after completion*.

A running thread t_1 invokes the method `join()` on a thread t_2. The `join()` call has no effect if thread t_2 has already completed. If thread t_2 is still alive, thread t_1 transits to one of the two waiting states. Depending on whether a timeout was specified or not in the call to the `join()` method, a call to the `getState()` method on thread t_1 while it is waiting for join completion will return the value `Thread.State.TIMED-_WAITING` or `Thread.State.WAITING`, respectively.

Thread t_1 waits in a waiting state until one of these events occurs (Figure 22.11):

- Thread t_2 completes.

  In this case, thread t_1 moves to the *READY* substate, and when it gets to run, it will continue normally after the call to the `join()` method.

- Thread t_1 is timed out.

  The time specified in the argument of the `join()` method call has elapsed without thread t_2 completing. In this case as well, thread t_1 transits from the `TIMED_WAITING` state to the *READY* substate. When it gets to run, it will continue normally after the call to the `join()` method.
  If the no-argument `join()` method was called, there is no

timeout for thread t_1 in the `WAITING` state. It waits indefinitely in this state for one of the other events to occur.

- Thread t_1 is interrupted.

  Some thread interrupted thread t_1 while it was waiting for join completion. Thread t_1 transits to the *READY* substate, but when it gets to execute, it will now receive an `InterruptedException`.

Figure 22.11 *Timed Joining of Threads*

Example 22.7 illustrates joining of threads. The `AnotherClient` class below uses the `Counter` class, which extends the `Thread` class, from Example 22.2. It creates two threads that are enabled for execution. The main thread invokes the `join()` method on the `Counter A` thread. If the `Counter A` thread has not already completed, the main thread transits to the `WAITING` state, as no timeout is specified. When the `Counter A` thread completes, the main thread will be enabled for running. Once the main thread is running, it continues with execution after (5). A parent thread can call the `isAlive()` method to find out whether its child threads are alive, before terminating itself. The call to the `isAlive()` method on the `Counter A` thread at (6) correctly reports that the `Counter A` thread is not alive. A similar scenario transpires between the main thread and the `Counter B` thread. At most, the main thread passes through the `WAITING` state twice.

Example 22.7 *Joining of Threads*

Click here to view code image

```
class Counter extends Thread { /* See Example 22.2, p. 1376. */ }
```

Click here to view code image

```
public class AnotherClient {
  public static void main(String[] args) {

    // Create two Counter threads, set their names, and start them:    // (4)
    Counter counterA = new Counter();
    Counter counterB = new Counter();
    counterA.setName("counterA");
    counterB.setName("counterB");
    counterA.start();
    counterB.start();

    try {
      System.out.println("Wait for the child threads to finish.");
      counterA.join();                                                  // (5)
      if (!counterA.isAlive()) {                                        // (6)
        System.out.println("Counter A not alive.");
      }
      counterB.join();                                                  // (7)
      if (!counterB.isAlive()) {
        System.out.println("Counter B not alive.");
      }
    } catch (InterruptedException ie) {
      System.out.println("main thread interrupted.");
    }
    System.out.println("Exiting from main thread.");
  }
}
```

Probable output from the program:

Click here to view code image

```
Wait for the child threads to finish.
counterB: 0
counterA: 0
counterA: 1
counterB: 1
counterA: 2
counterB: 2
counterA: 3
counterB: 3
counterA: 4
counterB: 4
Exiting counterB
Exiting counterA
Counter A not alive.
Counter B not alive.
Exiting from main thread.
```

Blocking for I/O

A running thread, on executing a *blocking operation* requiring a resource (like a call to an I/O method), will transit to the `BLOCKED` state (Figure 22.12). The `getState()` method on this thread will return the value `Thread.State.BLOCKED`. When the thread can complete the blocking operation, it proceeds to the *READY* substate. An example is a thread reading from the standard input terminal, that blocks until input is provided:

```
int input = System.in.read();
```

Figure 22.12 *Blocked for I/O*

Thread Termination

A thread can transit to the `TERMINATED` state from the `RUNNABLE` state (Figure 22.13). The thread terminates when it finishes executing its `run()` method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected. There is no way the thread can be enabled for running again, not even by calling the `start()` method again on the thread object. The `getState()` method on this thread will return the value `Thread.State.TERMINATED`.

Figure 22.13 *Thread Termination*

The *main thread* is special in that it is started and terminated when the `main()` method starts and stops execution, respectively.

It might be tempting to call the `System.exit()` method to terminate a thread, but this can be rather drastic. A call to this method terminates the *process* in which the JVM is running, and as a consequence, terminating all threads that might still be alive in the JVM.

Controlling a Thread

Example 22.8 illustrates a typical scenario where a thread can be controlled by one or more threads. Work is performed by a loop body, which the thread executes continuously. It should be possible for other threads to start and stop the *worker* thread. This functionality is implemented by the class `Worker` at (1), which has a private field `theThread` declared at (2) to keep track of the `Thread` object executing its `run()` method at (5).

The `kickStart()` method at (3) in the class `Worker` creates and starts a thread if the field `theThread` is not already denoting a running thread—that is, if the field has the `null` value. The `terminate()` method at (4) sets the field `theThread` to `null` to signal that the `run()` method can terminate, thus resulting in the thread being terminated. Note that this does not affect any `Thread` object that might have been referenced by the field `theThread`. The runtime system maintains any such `Thread` object; therefore, changing one of its references does not affect the `Thread` object.

The `run()` method at (5) has a loop whose execution is controlled by a special condition. The condition tests to see whether the `Thread` object referenced by the reference `theThread` and the `Thread` object currently running are one and the same. This is bound to be the case if the reference `theThread` has the same reference value that it was assigned when the thread was created and started in the `kickStart()` method. The condition will then be `true`, and the body of the loop will execute. However, if the value in the reference `theThread` has changed, the condition will be `false`. In that case, the loop will not execute, the `run()` method will complete, and the thread will terminate. This idiom is generally recommended to terminate a thread.

A client can control the thread implemented by the class `Worker`, using the `kick-Start()` and the `terminate()` methods. The client is able to terminate the running thread at the start of the next iteration of the loop body by calling the `terminate()` method that changes the value of the `theThread` reference to `null`.

In Example 22.8, a `Worker` object is first created at (8) and a thread started on this `Worker` object at (9). The main thread invokes the `sleep()` method at (10) to temporarily cease its execution for 2 milliseconds, giving the thread of the `Worker` object a chance to run. The main thread, when it is executing again, terminates the thread of the `Worker` object at (11), as explained earlier. This simple scenario can be generalized where several threads, sharing a single `Worker` object, could be starting and stopping the thread of the `Worker` object. However, this generalization also requires that the field `theThread` is declared `volatile` in order to avoid memory consistency errors (p. 1414).

Example 22.8 *Thread Termination*

Click here to view code image

```
public class Worker implements Runnable {                                 // (1)
  private Thread theThread;                                               // (2)

  public void kickStart() {                                               // (3)
    if (theThread == null) {
      theThread = new Thread(this);
      theThread.start();
    }
  }

  public void terminate() {                                               // (4)
    theThread = null;
  }

  @Override
  public void run() {                                                     // (5)
    while (theThread == Thread.currentThread()) {                         // (6)
      System.out.println("Going around in loops.");
    }
  }
}
```

Click here to view code image

```
public class Controller {
  public static void main(String[] args) {                                // (7)
    Worker worker = new Worker();                                         // (8)
    System.out.println("Start the worker.");
    worker.kickStart();                                                   // (9)
    try {
        Thread.sleep(2);                                    // (10)
      } catch(InterruptedException ie) {
        ie.printStackTrace();
      }
      System.out.println("Stop the worker.");
      worker.terminate();                                   // (11)
    }
  }
```

Probable output from the program:

```
Start the worker.
Going around in loops.
Going around in loops.
...
Going around in loops.
Going around in loops.
Stop the worker.
Going around in loops.
```

Deprecated Thread Methods

There are a few operations defined by the `Thread` class that are not recommended for use and are now marked as `@Deprecated`. These include the following methods of the `Thread` class:

> ```
> final void resume()
> final void stop()
> final void suspend()
> ```

Their initial design intention was to control the thread lifecycle. However, Java concurrency design has changed since these operations were introduced. The new concurrency API allows concurrent programming at a higher level than controlling the threads. The legacy approach to controlling threads was to force a thread to become suspended, to resume, or to stop. Deprecated concurrency methods are likely to result in memory corruption. The code in this chapter does not use any of the deprecated methods.

22.5 Thread Issues

In this section, we briefly outline some problems that occur often in multithreaded applications. Although some solutions are mentioned, there is no silver bullet that will solve all such problems. In the worst-case scenario, the application may need to be redesigned all over again.

Liveness and Fairness

Multithreaded applications strive for *liveness* and *fairness*.

The *fairness property* of a multithreaded application refers to threads in the application getting a fair chance to run—so that all threads in the application can make progress in their work, and no thread monopolizes the CPU at the expense of others.

The *liveness property* of a multithreaded application refers to the ability of the threads to execute in a *timely manner*—meaning performing as expected and making continuous progress in their work.

The rest of this section describes three liveness issues common in multithreaded applications: *deadlock*, *livelock*, and *starvation*.

Deadlock

A *deadlock* is a situation where threads are holding locks on objects that other threads need—a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to release a lock, they both remain waiting forever in a waiting state and never make any progress. The threads are said to be *deadlocked*.

A deadlock is depicted in Figure 22.14. Thread t_1 has a lock on object o_1, but cannot acquire the lock on object o_2. Thread t_2 has a lock on object o_2, but cannot acquire the lock on object o_1. They can only proceed if one of them releases a lock the other one wants, which is never going to happen.

Figure 22.14 *Deadlock*

The situation in Figure 22.14 is implemented in Example 22.9. Thread `t1` at (3) tries to synchronize at (4) and (5), first on string `o1` at (1), then on string `o2` at (2), respectively. The thread `t2` at (6) does the opposite. It synchronizes at (7) and (8), first on string `o2` and then on string `o1`, respectively. A deadlock can occur as explained previously, and the print statement at (11) will never be executed.

However, the potential of a deadlock in the situation in Example 22.9 is easy to fix. If the two threads acquire the locks on the objects in the same order, then mutual lock dependency is avoided and a deadlock can never occur. This means having the same locking order at (4) and (5) as at (7) and (8). In general, the cause of a deadlock is not always easy to discover, let alone easy to fix.

Example 22.9 *Deadlock*

Click here to view code image

```
public class DeadLockDanger {
  public static void main(String[] args) {
    String o1 = "o1 " ;                                           // (1)
    String o2 = "o2 ";                                            // (2)

    Thread t1 = (new Thread("t1") {                               // (3)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o1) {                                      // (4)
            System.out.println(threadName + " acquired " + o1);
            synchronized(o2) {                                    // (5)
              System.out.println(threadName + ": " + o1 + o2);
      }}}}});

    Thread t2 = (new Thread("t2") {                               // (6)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o2) {                                      // (7)
            System.out.println(threadName + " acquired " + o2);
            synchronized(o1) {                                    // (8)
              System.out.println(threadName + ": " + o2 + o1);
      }}}}});

    t1.start();                                                   // (9)
    t2.start();                                                   // (10)
    System.out.println("Exiting main.");                          // (11)
  }
}
```

It is possible that the program in Example 22.9 might run without a deadlock, depending on how the locks on the two `String` objects are acquired by each thread. However, after the following output from the program, a deadlock occurs and the program never terminates:

```
t1 acquired o1
t2 acquired o2
```

Livelock

Two threads may result in a livelock, if the response by a thread to another thread's action is always to undo or revert the consequences of the action. The threads are responding to each other, but are blocked in making any progress. The combined actions of the threads make it impossible for them to complete their tasks, making it impossible for the threads to terminate. This situation is different from a deadlock, where the threads are blocked, waiting for locked shared resources to become available.

An example of a livelock is when two guests (i.e., two threads) are trying to enter through a door, but there is only one door key and only one guest at a time can use it. Each guest is too polite, and insists on giving the key to the other if the other wants to enter. In other words, if one guest hands the key to the other, the other hands it right back. This situation will continue indefinitely, with neither guest thereby entering through the door. We have a livelock because of handing the key back and forth.

Example 22.10 provides another example of a livelock. Two threads are created at (11) and (12). In the first one, a customer makes a payment to a seller, and in the second one, the seller ships an item to the customer.

The `Customer` class at (1) has a flag (`paymentMade`) to mark that payment has been made. In the `makePaymentTo()` method, the customer continuously checks whether the seller has sent the shipment. If the seller has not, the customer waits before checking again. Once the seller has shipped the item, the customer marks at (4) that the payment has now been made.

On the other hand, the `Seller` class at (6) has a flag (`itemShipped`) to record that shipment has been sent. In the `shipTo()` method, the seller continuously checks whether the customer has paid. If the customer has not, the seller waits before checking again. Once the customer has paid, the seller marks at (5) that the shipment has been sent.

Output from running the program shows that the customer is waiting for shipment from the seller, and the seller is waiting for payment from the customer. Neither thread is able to make progress because the seller cannot send the shipment before the customer has paid, and the customer cannot pay before the shipment has been sent. Each is expecting a confirmation from the other, which never arrives. They need to renegotiate their contract, and one of them will have to trust the other.

Example 22.10 *Livelock*

Click here to view code image

```
public class Customer {                                                  // (1
  private boolean paymentMade = false;                                   // (2

  public void makePaymentTo(Seller seller) {                             // (3
    while (!seller.hasShipped()) {
      System.out.println("Customer: waiting for shipment from seller");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Customer: payment made");
    this.paymentMade = true;                                              // (4
  }

  public boolean hasPaid() {
    return this.paymentMade;
  }
}
```

Click here to view code image

```
public class Seller {                                                     // (5
  private boolean itemShipped = false;                                    // (6

  public void shipTo(Customer customer) {                                 // (7
    while (!customer.hasPaid()) {
      System.out.println("Seller: waiting for payment from customer");
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Seller: item shipped");
    this.itemShipped = true;                                              // (8
  }

  public boolean hasShipped() {
    return this.itemShipped;
  }
}
```

Click here to view code image

```
public class LivelockShipment {
  public static void main(String[] args) {
    Customer customer = new Customer();
    Seller seller = new Seller();

    new Thread(() -> customer.makePaymentTo(seller)).start();          // (9
    new Thread(() -> seller.shipTo(customer)).start();                 // (1
  }
}
```

Output from the program:

Click here to view code image

```
...
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
...
```

Starvation

A thread can *starve* because it is unsuccessfully waiting for its turn to be able to proceed with its execution. In addition to deadlock and livelock that result in starvation, some other situations where starvation can occur are the following:

- *Unfair priority scheduling*: Higher-priority threads monopolize the CPU, and low-priority threads starve in the *READY* substate.
- *Indefinite waiting for join to complete:* A thread that executed a no-timeout `join()` call can starve in the `WAITING` state if the thread it is waiting for never completes.
- *Indefinite waiting in the entry set of the object lock:* A thread can starve in the `BLOCKED` state if other threads get chosen before it to acquire the object lock to enter synchronized code.
- *Indefinite waiting in the wait set of an object*: A thread that executed a no-timeout `wait()` call can starve in the `WAITING` state if it never gets notified. It can also starve in this state if other threads get chosen on notification before it does, and it never gets to compete for object lock acquisition.

Example 22.11 provides another example of starvation. The `Hole` class at (1) provides the `synchronized` method `dig()` to dig a hole. The class `Diggers` creates a `Hole` object and starts five threads to dig this hole. The output shows that the thread that first acquires the lock to execute the `synchronized dig()` method in the `Hole` object monopolizes the CPU, and the other threads do not get a chance to run—they wait and starve.

The starvation scenario in Example 22.11 can be remedied by making a thread wait between digging—implemented by the `try-catch` block at (3) with the call to the *timed* `wait()` method. The call to the `wait()` method will result in the current thread transiting to the `TIMED_WAITING` state and releasing the lock on the `Hole` object, thus making it possible for other waiting threads to run. On the other hand, a call to the `sleep()` method does not release the lock and will still result in thread starvation.

Example 22.11 *Starvation*

Click here to view code image

```
public class Hole {                                                  // (1)
  public synchronized void dig() {
    String threadName = Thread.currentThread().getName();
    while (true) {                                                   // (2)
```

```
          System.out.println(threadName + " digging the hole.");
//        try {                                                       // (3)
//        wait(1);
//        } catch (InterruptedException ex) {
//          ex.printStackTrace();
//        }
        }
      }
    }
```

Click here to view code image

```
public class Diggers {
  public static void main(String[] args) {
    Hole hole = new Hole();                                         // (4)
    for (int i = 0; i < 5; i++) {                                   // (5)
      new Thread(() -> hole.dig()).start();
    }
  }
}
```

Probable output from the program:

```
...
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
Thread-0 digging the hole.
...
```

Memory Consistency Errors

In a multithreaded application, there is a potential danger that changes made by one thread to shared data might not be *visible* to the other threads, resulting in inconsistent views of the shared data. When this happens, it is known as a *memory consistency error*.

Causes of memory consistency errors are too complex to go into here. The following simple example gives an idea of how they can manifest. Two threads have access to a shared counter.

Click here to view code image

```
int counter = 0;

// Thread A                                     // Thread B
...                                             ...
counter = 1;                                    ...
System.out.println("Thread A:" + counter);      System.out.println("Thread B:"
...                                             ...
```

Thread `A` set the counter value to `1` and prints it value. We can safely assume that the value printed would be `1`—that is, the value in the counter set by the preceding assignment statement. However, if the print statement is executed by thread `B`, there is no guarantee that the change in the counter value in thread `A` will be visible to thread `B`—it might print `0` or `1`—unless it can be established that a change of value in thread `A` occurred before being printed in thread `B`.

Happens-Before Relationship

The *happens-before relationship* helps to combat memory consistency errors. It is important in guaranteeing that *one action is ordered before another* in a multithreaded runtime environment. More importantly, the first action is *visible* to the second one, meaning that the *results* of the first action are evident to the second action. The relationship is *transitive*: If action `A` happens-before action `B`, and action `B` happens-before action `C`, then action `A` happens-before action `C`.

Little is guaranteed in a multithreaded environment, and therefore, the happens-before relationship is important in ascertaining certain properties about multi-threaded applications. Here are some important rules pertaining to multithreaded applications in Java:

- *Object lock rule*: An unlock action on a monitor happens-before every subsequent lock action on that monitor.
- *Volatile field rule*: A write to a `volatile` field happens-before every subsequent read of that field (§23.4, p. 1453).
- *Thread start rule*: A call to the `Thread.start()` method on a thread happens-before any actions in the started thread.
- *Thread termination rule*: All actions in a thread happen-before any other thread can detect that the thread has terminated, either by successfully returning from a `Thread.join()` method call or by a `Thread.isAlive()` method call returning `false` on that thread.
- *Thread interruption rule*: A thread calling the `Thread.interrupt()` method on another thread happens-before the interrupted thread detects the interrupt, either by having an `InterruptedException` thrown, or by invoking the `Thread .isInterrupted()` or the `Thread.interrupted()` method on itself.

Review Questions

22.4 Which of the following statements are guaranteed to be true about the following program?

Click here to view code image

```
public class ThreadedPrint {
  static Thread makeThread(String id, boolean daemon) {
    Thread t = new Thread(() -> System.out.println(id), id);
    t.setDaemon(daemon);
    t.start();
    return t;
  }

  public static void main(String[] args) {
    Thread a = makeThread("A", false);
    Thread b = makeThread("B", true);
    System.out.println("End");
  }
}
```

Select the two correct answers.

- **a.** The letter `A` is always printed.
- **b.** The letter `B` is always printed.
- **c.** The letter `A` is never printed after `End`.
- **d.** The letter `B` is never printed after `End`.
- **e.** The program might print `B`, `End`, and `A`, in that order.

22.5 Which of the following statements are true about the `synchronized` statement? Select the two correct answers.

- **a.** If the expression in a `synchronized` statement evaluates to `null`, a `NullPointer-Exception` will be thrown.
- **b.** The lock is only released if the execution of the `synchronized` statement terminates normally.
- **c.** Several `synchronized` statements, synchronizing on the same object, can be executed concurrently.
- **d.** Synchronized statements cannot be nested.
- **e.** The block notation, `{}`, cannot be omitted even if there is only a single statement to execute in a `synchronized` statement.

22.6 Given the following program, which code modifications will result in *both* threads being able to participate in printing one smiley face (`:-)`) on a line continuously?

Click here to view code image

```
public class Smiley extends Thread {

  public void run() {                                              // (1
    while (true) {                                                 // (2
      try {                                                        // (3
          System.out.print(":");                                   // (4
          sleep(100);                                              // (5
          System.out.print("-");                                   // (6
          sleep(100);                                              // (7
          System.out.println(")");                                 // (8
          sleep(100);                                              // (9
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  public static void main(String[] args) {
    new Smiley().start();
    new Smiley().start();
  }
}
```

Select the two correct answers.

a. Synchronize the `run()` method at (1) with the keyword `synchronized`.

b. Synchronize the `while` loop at (2) with a `synchronized(Smiley.class)` statement.

c. Synchronize the `try-catch` construct at (3) with a `synchronized(Smiley.class)` statement.

d. Synchronize the statements from (4) to (9) with a `synchronized(Smiley.class)` statement.

e. Synchronize each statement at (4), (6), and (8) individually with a `synchronized (Smiley.class)` statement.

22.7 Which of the following events will cause a thread to terminate?

Select the one correct answer.

a. The method `sleep()` is called.

b. The method `wait()` is called.

c. Execution of the `start()` method ends.

d. Execution of the `run()` method ends.

e. Execution of the thread constructor ends.

22.8 Which of the following statements are true about the following code?

Click here to view code image

```
public class Joining {
  static Thread createThread(final int i, final Thread t1) {
    Thread t2 = new Thread() {
      public void run() {
          System.out.println(i+1);
          try {
            t1.join();
          } catch (InterruptedException ie) {
          }
          System.out.println(i+2);
        }
      };
      System.out.println(i+3);
      t2.start();
      System.out.println(i+4);
      return t2;
    }

    public static void main(String[] args) {
      createThread(10, createThread(20, Thread.currentThread()));
    }
  }
```

Select the two correct answers.

a. The first number printed is `13`.
b. The number `14` is printed before the number `22`.
c. The number `24` is printed before the number `21`.
d. The last number printed is `12`.
e. The number `11` is printed before the number `23`.

22.9 What can be guaranteed by calling the method `yield()`?

Select the one correct answer.

a. All lower-priority threads will be granted CPU time.
b. The current thread will sleep for some time while some other threads run.
c. The current thread will not continue until other threads have terminated.
d. The thread will wait until it is notified.
e. None of the above

22.10 In which class or interface is the `notify()` method defined?

Select the one correct answer.

a. `Thread`
b. `Object`
c. `Appendable`
d. `Runnable`

22.11 What will be the result of invoking the `wait()` method on an object without ensuring that the current thread holds the lock on the object?

Select the one correct answer.

a. The code will fail to compile.
b. Nothing special will happen.

c. An `IllegalMonitorStateException` will be thrown if the `wait()` method is called and the current thread does not hold the lock of the object.

d. The thread will be blocked until it gains the lock of the object.

22.12 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Tank {
  private boolean isEmpty = true;

  public synchronized void emptying() {
    pause(true);
    isEmpty = !isEmpty;
    System.out.println("emptying");
    notify();
  }

  public synchronized void filling() {
    pause(false);
    isEmpty = !isEmpty;
    System.out.println("filling");
    notify();
  }

  private void pause(boolean flag) {
    while (flag ? isEmpty : !isEmpty) {
      try {
        wait();
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread().getName() + " interrup
      }
    }
  }

  public static void main(String[] args) {
    final Tank token = new Tank();
    new Thread(() -> {for(;;) token.emptying();}, "A").start();
    new Thread(() -> {for(;;) token.filling();}, "B").start();
  }
}
```

Select the one correct answer.

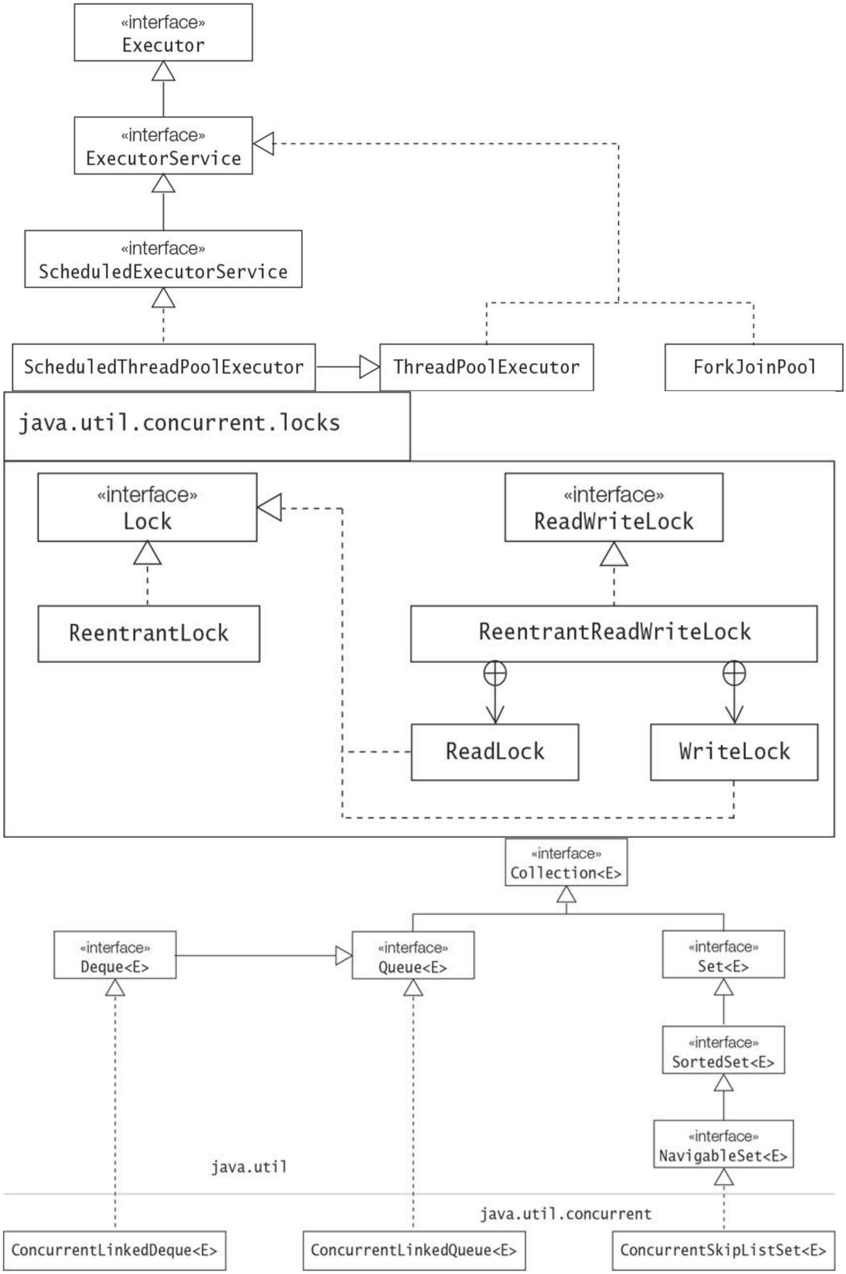
a. The program will compile and continue running once started, but will not print anything.

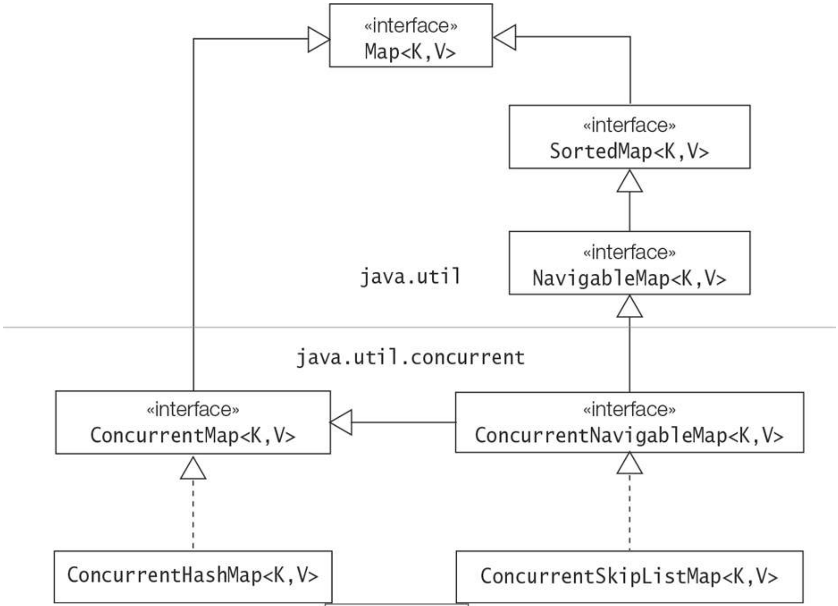
b. The program will compile and continue running once started, printing only the string `"emptying"`.

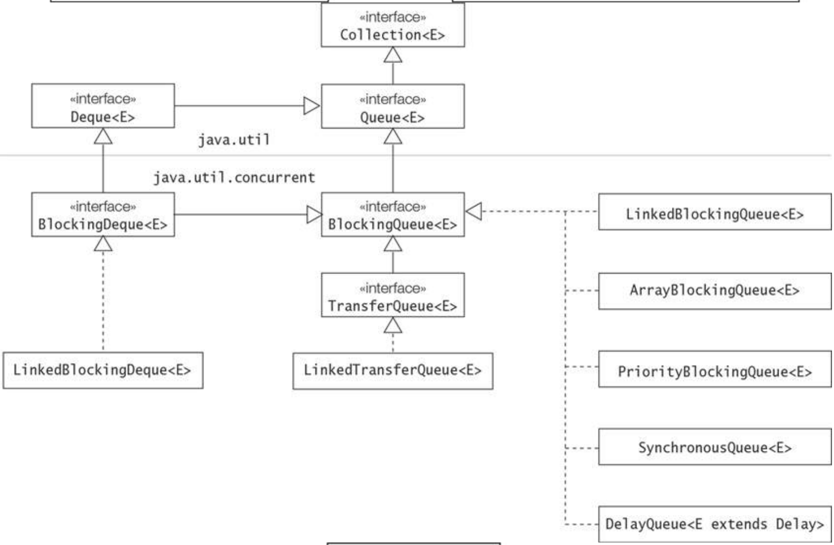
c. The program will compile and continue running once started, printing only the string `"filling"`.

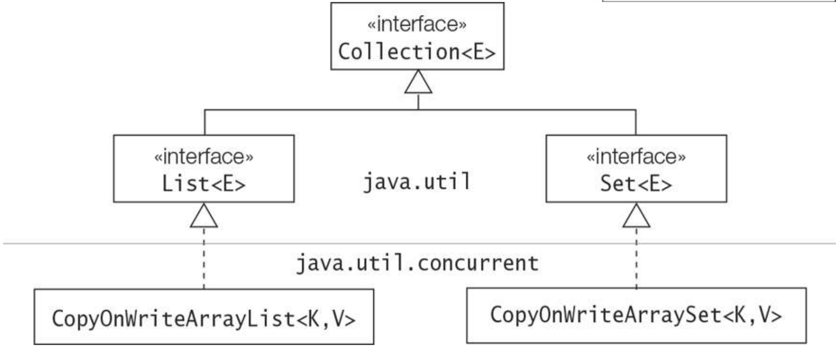
d. The program will compile and continue running once started, always printing the string `"filling"` followed by the string `"emptying"`.

e. The program will compile and continue running once started, printing the strings `"filling"` and `"emptying"` in some order.


«interface»
Executor
«interface»
ExecutorService
«interface»
ScheduledExecutorService
ScheduledThreadPoolExecutor
ThreadPoolExecutor
ForkJoinPool
java.util.concurrent.locks
«interface»
Lock
ReentrantLock
«interface»
ReadWriteLock
ReentrantReadWriteLock
ReadLock
WriteLock
«interface»
Collection<E>
«interface»
Deque<E>
«interface»
Queue<E>
«interface»
Set<E>
«interface»
SortedSet<E>
«interface»
NavigableSet<E>
java.util
java.util.concurrent
ConcurrentLinkedDeque<E>
ConcurrentLinkedQueue<E>
ConcurrentSkipListSet<E>


«interface»
Map<K,V>
«interface»
SortedMap<K,V>
«interface»
NavigableMap<K,V>
java.util
java.util.concurrent
«interface»
ConcurrentMap<K,V>
«interface»
ConcurrentNavigableMap<K,V>
ConcurrentHashMap<K,V>
ConcurrentSkipListMap<K,V>


«interface»
Collection<E>
«interface»
Deque<E>
«interface»
Queue<E>
java.util
java.util.concurrent
«interface»
BlockingDeque<E>
«interface»
BlockingQueue<E>
LinkedBlockingQueue<E>
ArrayBlockingQueue<E>
«interface»
TransferQueue<E>
LinkedBlockingDeque<E>
LinkedTransferQueue<E>
PriorityBlockingQueue<E>
SynchronousQueue<E>
DelayQueue<E extends Delay>


«interface»
Collection<E>
«interface»
List<E>
java.util
«interface»
Set<E>
java.util.concurrent
CopyOnWriteArrayList<K,V>
CopyOnWriteArraySet<K,V>

Concurrency: Part II 23

Chapter Topics

- Understanding the inheritance hierarchy of the main interfaces of the Executor Framework: `Executor`, `ExecutorService`, and `ScheduledExecutorService` interfaces
- Using time units for time durations defined by the `TimeUnit` enum type
- Using a thread-local pseudorandom number generator defined by the `ThreadLocalRandom` class
- Creating executor services and scheduled executor services using methods of the `Executors` utility class
- Understanding the lifecycle of an executor service: `RUNNING`, `SHUTDOWN`, and `TERMINATION` states
- Controlling shutdown and termination of an executor service
- Defining tasks that do not return a result and those that do using the `Runnable` and the `Callable<V>` functional interfaces, respectively
- Submitting and invoking tasks using an executor service
- Handling task results using the `Future<V>` interface
- Scheduling tasks with a scheduled executor service
- Understanding task cancellation in an executor service
- Understanding the basics of solving divide-and-conquer problems using the Fork/Join Framework

- Understanding the implications of immutability
- Using the keyword `volatile` for visibility of shared fields to avoid memory contention errors
- Using atomic variables for mutually exclusive operations on shared single fields
- Using programmatic locking provided by reentrant locks and reentrant read-write locks to implement mutually exclusive operations on shared resources
- Using special-purpose synchronizers, such as the cyclic barrier and the count-down latch, to control thread execution behavior
- Using synchronized collections and maps for mutually exclusive operations on collections
- Using concurrent collections, concurrent maps, blocking queues, and copy-on-write collections for mutually exclusive operations on collections and maps

Java SE 17 Developer Exam O

§23.2, p. 1423

[8.1] Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different Executor services and concurrent API

- ○ *Creating worker threads with the `Callable` functional interface, and managing concurrency using executor services of the Concurrency API are covered in this chapter.*
- ○ *For the `Runnable` functional interface and the thread lifecycle, see §22.3, p. 1371, and §22.4, p. 1380.*

§23.4, p. 1451 to §23.7, p. 1482

[8.2] Develop thread-safe code, using different locking mechanisms and concurrent API

§23.7, p. 1482

[8.3] Process Java collections concurrently including the use of parallel streams

- ○ *Processing collections concurrently is covered in this chapter.*
- ○ *For parallel streams, see §16.9, p. 1009.*

Java SE 11 Developer Exam O

§23.2, p. 1423

[8.1] Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API

- ○ *Creating worker threads with the `Callable` functional interface, and managing concurrency using executor services* *of the Concurrency API are covered in this chapter.*

For the `Runnable` *functional interface, see §22.3, p. 1371.*

§23.4, p. 1451 to §23.7, p. 1482

[8.2.1] Develop thread-safe code, using different locking mechanisms and java.util.concurrent API

Low-level support for creating, starting, and coordinating threads in multithreaded applications is discussed in Chapter 22, p. 1365. This low-level approach to enforce correct execution order for concurrent threads inherently increases code complexity, potentially impacts performance, and is not always scalable. This chapter deals with high-level support for multithreaded programming that alleviates many of these problems in designing multithreaded applications. This support is primarily provided by the `java.util.concurrent` package and its subpackages: The Executor Framework, the Fork/Join Framework, locking mechanisms, lock-free algorithms, and concurrent collections.

23.1 Utility Classes `TimeUnit` and `ThreadLocalRandom`

This section can be skipped on the first reading, and can be read if and when the need arises while working through this chapter.

Before diving into high-level concurrency support provided by the `java.util.concurrent` package, we begin by providing an overview of two utility classes whose methods are used in many of the examples presented in this chapter.

The `TimeUnit` Enum Type

The enum type `java.util.concurrent.TimeUnit` represents *time units* at different levels of granularity, from nanoseconds to days, as shown in Table 23.1.

A `TimeUnit` can be used to indicate the time unit in which a numerical value should be interpreted as a time duration. Many time-based methods interpret their timing parameter according to the time unit specified by this enum type. In the code below, the numerical value 2 will be interpreted by the `awaitTermination()` method as two seconds.

Click here to view code image

```
boolean allTerminated = executor.awaitTermination(2, TimeUnit.SECONDS);
```

Note that a `TimeUnit` enum constant does not maintain any time information, like date or time of day.

Table 23.1 *Constants Defined by the* `java.util.concurrent.TimeUnit` *Enum Type*

Constant defined by java.util type	**Description of the time unit**
`NANOSECONDS`	Representing one-thousandth of a microsecond
`MICROSECONDS`	Representing one-thousandth of a millisecond
`MILLISECONDS`	Representing one-thousandth of a second
`MINUTES`	Representing 60 seconds
`SECONDS`	Representing 1 second
`HOURS`	Representing 60 minutes
`DAYS`	Representing 24 hours

The enum type `TimeUnit` also provides convenience methods that call the methods `Thread.sleep()`, `Thread.join()`, and `Object.wait()`, allowing the timeout for these methods to be specified in different time units. For example, the following method calls are equivalent:

Click here to view code image

```
TimeUnit.SECONDS.sleep(1);              // 1 second.
TimeUnit.MILLISECONDS.sleep(1000);      // 1000 milliseconds.
Thread.sleep(1000);                     // 1000 milliseconds.
```

Following convenience methods are defined in the `TimeUnit` enum type:

> Click here to view code image
>
> ```
> void sleep(long timeout)                       throws InterruptedException
> void timedJoin(Thread thread, long timeout)    throws InterruptedException
> void timedWait(Object obj, long timeout)       throws InterruptedException
> ```
>
> Call the `Thread.sleep(long millis)` method, the `Thread.join(long millis)` method, and the `Object.wait(long millis)` method, respectively, where the specified `timeout` is converted to milliseconds using this time unit.

The `ThreadLocalRandom` Class

Some examples in this chapter make use of a *pseudorandom generator*. The class `java.util.concurrent.ThreadLocalRandom` provides a pseudorandom generator that is confined to the current thread—that is, it is local to the thread. The `ThreadLocalRandom` class extends the `java.util.Random` class, and provides methods that return uniformly distributed numerical values. The `ThreadLocalRandom` class is particularly recommended for concurrent tasks, instead of using a shared object of the `Random` class, as it entails less overhead and resource contention.

The code below illustrates use of the class in a task that simulates a dice. The `current()` static method of the class returns the `ThreadLocalRandom` object associated with the current thread.

Click here to view code image

```
int diceValue = ThreadLocalRandom.current().nextInt(1, 7); // [1, 6]
```

In addition, the `ThreadLocalRandom` class provides methods that return numerical streams whose elements are uniformly distributed pseudorandom numbers. The code below creates an `int` stream of 100 randomly generated `int` values which are in the interval `[1, 6]`, and counts the number of times the dice value `6` occurs in the stream.

Click here to view code image

```
long count = ThreadLocalRandom.current()
                              .ints(100, 1, 7).filter(i -> i == 6).count();
```

Following selected methods are defined in the `ThreadLocalRandom` enum type:

> Click here to view code image
>
> ```
> static ThreadLocalRandom current()
> ```
>
> Returns the `ThreadLocalRandom` object associated with the current thread.
>
> Click here to view code image
>
> ```
> int nextInt(int bound)
> int nextInt(int origin, int bound)
> ```
>
> The two methods return a pseudorandom, uniformly distributed `int` value between `[0, bound)` and `[origin, bound)`, respectively, where `[n, m)` defines a half-open interval, where the end value is excluded.
> The class `ThreadLocalRandom` also defines the analogous methods `nextLong()`, `nextFloat()`, and `nextDouble()` for the primitive types `long`, `float`, and `double`, respectively.

Click here to view code image

```
IntStream ints()
IntStream ints(long streamSize)
IntStream ints(int randomNumberOrigin, int randomNumberBound)
IntStream ints(long streamSize, int randomNumberOrigin,
               int randomNumberBound)
```

Each method returns a stream of pseudorandom `int` values.
The first method returns an unlimited number of pseudorandom `int` values in the stream, which can be any legal `int` value.
The second method returns `streamSize` pseudorandom `int` values in the stream, which can be any legal `int` value.
The third method returns unlimited pseudorandom `int` values in the stream, which are in the half-open interval `[randomNumberOrigin, randomNumberBound)`.
The fourth method returns `streamSize` pseudorandom `int` values in the stream, which are in the half-open interval `[randomNumberOrigin, randomNumberBound)`.
The class `ThreadLocalRandom` also defines the analogous methods `longs()` and `doubles()` for the primitive types `long` and `double` that return `LongStream` and `DoubleStream`, respectively.

23.2 The Executor Framework

Executors provide a high-level approach to launching tasks and managing threads. Tasks are executed asynchronously by threads in a multithreaded environment, where a *task* defines a unit of work—that is, a task is a set of instructions executed by a thread.

Internally, an executor maintains a *thread pool* that utilizes a number of reusable threads that are assigned tasks from a task queue, according to the execution policy of the executor. The threads maintained by a thread pool are also known as *worker threads*. A worker thread remains alive after executing a task, and can be assigned another task that is waiting to be executed.

Executors allow the *submission of tasks* for execution to be decoupled from the actual *execution of tasks*. The application defines and submits the tasks to the executor which manages their execution by assigning them to threads. The executor provides the necessary support for managing its lifecycle that comprises creating the executor, submitting tasks, managing the outcome of task execution, and shutting down the executor.

Figure 23.1 shows the interfaces in the `java.util.concurrent` package that define the characteristics of executors from basic to more advanced executors. The `Executors` utility class provides static methods that create various kinds of executors that not only implement the executor interfaces shown in Figure 23.1, but also implement various policies for task execution (Table 23.2, p. 1426).

Figure 23.1 *Executor Interfaces in the* `java.util.concurrent` *Package*

The `Executor` Interface

The `java.util.concurrent.Executor` interface defines a basic executor. This executor executes a `Runnable` task that is passed to its `execute()` method. Technically, the `Executor` interface is a functional interface as it has a single abstract method. However, it is not annotated as such in the Java SE Platform API documentation, as it is recommended to use the more versatile executors that extend the `Executor` interface (Figure 23.1).

The `SimpleExecutor` class below implements the `Executor` interface. Its `execute()` method shows two implementations, (1a) and (1b), for executing a `Runnable` task. Typically, an executor delegates the task to a new thread for asynchronous execution as shown at (1a). It can also be executed synchronously by calling the `run()` method on the task as shown at (1b).

Click here to view code image

```
class SimpleExecutor implements Executor {
  @Override
  public void execute(Runnable task) {

    new Thread(task).start();         // (1a) Asynchronous call
//  task.run();                       // (1b) Synchronous call. Not recommende
  }
}
// Client code
SimpleExecutor executor = new SimpleExecutor();
Runnable task = () -> System.out.println("Executing task ...");
executor.execute(task);
```

The essence of the basic executor is to replace the following code for running a task in a thread:

```
new Thread(task).start();
```

with the following code using an executor, where the actual execution is managed by the executor, abstracting away the management of the underlying thread:

```
executor.execute(task);
```

Typically, the tasks are executed by a *thread pool* maintained by the executors provided by the `Executors` utility class.

The following method is defined in the `Executor` interface:

> ```
> void execute(Runnable task)
> ```
>
> Executes the given `task` at some time in the future. The task may execute in a new thread, in a worker thread, or in the calling thread, at the discretion of the `Executor` implementation. Note that this method does not return a value and it does not specify any `throws` clause with checked exceptions.

The `Executors` Class

The `java.util.concurrent.Executors` class provides factory and utility methods for the interfaces defined in the `java.util.concurrent` package. In particular, it provides factory methods that return versatile implementations of the `ExecutorService` (p. 1427) and the `ScheduledExecutorService` (p. 1440) interfaces shown in Figure 23.1. The executor services created by the methods of the `Executors` class are summarized in Table 23.2, showing the respective static methods, the task execution policy of the executor services they create, and examples where they are used. For many applications, these executor services will be adequate, and examples in this chapter make heavy use of them.

In addition, the `java.util.concurrent.ThreadPoolExecutor` class, shown in Figure 23.1, provides constructors to create even more versatile executors with various properties such as minimum and maximum pool size, keep-alive time for idle threads, a blocking queue for managing tasks, and a handler for tasks that cannot be executed.

Table 23.2 *Selected Executor Services Provided by the* `Executors` *Utility Class*

The static method that creates `ExecutorService` implementation	**Description of the `ExecutorService` interface (p. 1427), and the executor's task execution policy**
SingleThreadExecutor	Provides a single worker thread to which tasks can be assigned. If

`newSingleThreadExecutor()` *Task execution policy*: tasks executed sequentially by a single thread (*Example 23.5, p. 1446*)	this single thread terminates due to a failure, a new thread will take its place if needed. Tasks in the queue are guaranteed to execute sequentially. Logically it is an equivalent of a fixed thread pool of size 1, but it cannot be reconfigured to use additional threads.
Fixed Thread Pool `newFixedThreadPool(int nThreads)` *Task execution policy*: tasks executed concurrently by a fixed number of threads. (*Example 23.1, p. 1429*) (*Example 23.2, p. 1438*)	Provides a thread pool of fixed size. The number of threads actively processing tasks is, at the most, equal to the pool size. The number of tasks submitted can be greater than the pool size, but some tasks may have to wait in the queue for threads to become available within the pool. If a thread terminates due to a failure, a new thread will take its place if needed. Pool threads are reused as required to execute the submitted tasks. They remain blocked when not executing a task.

Work Stealing Pool
Click here to view code image
`newWorkStealingPool()`
`newWorkStealingPool(int parallelism)`
Task execution policy: tasks executed concurrently while attempting to maintain its target parallelism level.
	Provides a thread pool that tries to maintain its *target parallelism level*—that is, the maximum number of threads executing, or those available to execute tasks. The actual number of threads can vary dynamically. It makes no guarantees about the order in which the tasks will be executed. The first method uses the number of available processors as its target parallelism level.

Cached Thread Pool `newCachedThreadPool()` *Task execution policy*: tasks executed concurrently by a thread pool whose size can vary dynamically with the number of tasks remaining to execute.	Provides a thread pool whose size may vary dynamically. Submitted tasks reuse available threads in the pool. Additional threads are created as needed when no threads are available. Unused threads eventually time out and are removed from the cache, so a pool that remains idle for long enough will not consume any resources.
Single Thread Scheduled Executor Click here to view code image `newSingleThreadScheduledExecutor()` *Task execution policy*: tasks executed sequentially by a single thread according to the schedule policy specified for each task. (*Example 23.5, p. 1446*)	Provides a single-thread executor that works very much like a combination of a scheduled thread pool of size 1 and a single thread executor. It can execute submitted tasks sequentially, using a single thread, but with a specified delay, or periodically, based on a schedule. The schedule is specified individually for each task when it is submitted to the executor.

Scheduled Thread Pool
`newScheduledThreadPool(`
`int corePoolSize)`
Task execution policy: tasks executed concurrently by a fixed number of threads, according to the schedule policy specified for each task.
(*Example 23.3, p. 1441*)
(*Example 23.5, p. 1446*)
	Provides a thread pool of a fixed size, that can execute submitted tasks with a specified delay or periodically, based on a schedule. The schedule is specified individually for each task when it is submitted to the executor.

The `ExecutorService` Interface

Table 23.2 lists the executor services provided by the factory methods of the `Executors` class. An *executor service* implements the `ExecutorService` interface that extends the `Executor` interface with methods that provide the following functionality:

- Flexibly submitting tasks to the executor service and handling the results returned (p. 1436)

  In addition to the `execute()` method, an executor service provides the overloaded methods `submit()`, `invokeAll()`, and `invokeAny()` for submitting tasks. The `Future<V>` interface provides methods to handle the execution status and the results returned by tasks (p. 1435). The `Callable<V>` functional interface can be used to implement tasks that return a value (p. 1434).

- Managing the *executor lifecycle*

  The `ExecutorService` interface provides methods to manage the lifecycle of an executor (Figure 23.2). Once an executor service is created, for example, by static methods of the `Executors` class, it is in the `RUNNING` state, ready to accept tasks for execution. As an executor service is a precious resource, it must be shut down by calling its `shutdown()` or `shutdownNow()` method. An executor service in the `SHUTDOWN` state does not accept new tasks, which are discarded. However, although the executor service might be shut down, some of its threads might still be executing tasks and other tasks might be waiting to be executed. Only when all tasks have completed does the executor service transition to the `TERMINATED` state. The methods `isShutdown()` and `isTerminated()` can be used to check if the executor service is in the `SHUTDOWN` or `TERMINATED` state, respectively. The method `awaitTermination()`, called on an executor service after it is shut down, can be used to wait for threads to complete their tasks. See Example 23.1, p. 1429.

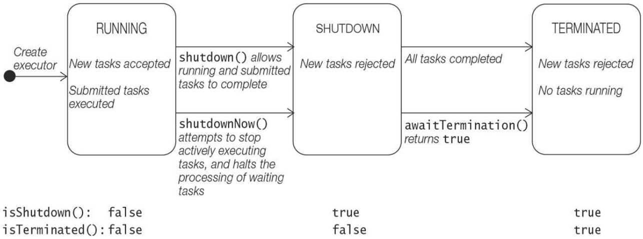

RUNNING
Create executor
New tasks accepted
Submitted tasks executed
shutdown() allows running and submitted tasks to complete
shutdownNow() attempts to stop actively executing tasks, and halts the processing of waiting tasks
SHUTDOWN
New tasks rejected
All tasks completed
awaitTermination() returns true
TERMINATED
New tasks rejected
No tasks running
isShutdown(): false
isTerminated(): false
true
false
true
true

Figure 23.2 *Executor Service Lifecycle*

Following are selected methods of the `ExecutorService` interface for shutting down and terminating an executor service:

> ```
> void shutdown()
> ```
>
> Starts an orderly shutdown in which previously submitted tasks are executed, but new tasks are rejected. This method does *not* wait for already submitted tasks to complete execution. A shutdown request on an executor that is already shut down has no additional effect.
>
> ```
> List<Runnable> shutdownNow()
> ```
>
> Attempts to stop all actively executing tasks, and halts the processing of waiting tasks. It returns a list of the tasks that were awaiting execution.
>
> This method does *not* wait for actively executing tasks to terminate.
>
> This method at best attempts to stop processing of actively executing tasks— however, there are no guarantees. Typically, task cancellation will be triggered via `Thread.interrupt()`, so any task failing to respond to interrupts may never terminate (§22.4, p. 1393).
>
> ```
> boolean isShutdown()
> ```
>
> Returns `true` if this executor has been shut down—that is, a shutdown request has already been invoked on the executor.
>
> Click here to view code image
>
> ```
> boolean awaitTermination(long timeout, TimeUnit unit)
>                          throws InterruptedException
> ```
>
> Blocks until one of these events occurs first: All tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted. This method should be called after calling the `shutdown()` method.
>
> ```
> boolean isTerminated()
> ```
>
> Returns `true` if *all* tasks have completed following shutdown. It never returns `true` unless a shutdown request has already been invoked on the executor.

Using an Executor Service

The idiom for using an executor service is embodied in the following steps that are illustrated in Example 23.1:

- *Create the executor service.*
- *Submit tasks to the executor service.*
- *Shut down and terminate the executor service.*

Example 23.1 *Executor Lifecycle*

Click here to view code image

```
package executors;
import java.util.concurrent.*;

public class ExecutorLifecycle {

  // Task: dice roll.
  private static final Runnable diceRoll = () -> {                          //
    int diceValue = ThreadLocalRandom.current().nextInt(1, 7);           // [1,

    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + " => dice value: " + diceValue);
    try {
      TimeUnit.MILLISECONDS.sleep(100);
    } catch (InterruptedException e) {
      System.out.println(threadName + ": " + e);
    }
  };

  public static void main(String[] args) {
    System.out.printf("%50s   %s%n", "isShutdown()", "isTerminated()");
    // Create the executor service:
    ExecutorService es = Executors.newFixedThreadPool(3);                   //
    try {                                                                   //
      checkStates(es, "Before execute() at (4): ");

      // Submit tasks:
      es.execute(diceRoll);                                                 //
      es.execute(diceRoll);
      es.execute(diceRoll);
      checkStates(es, "After execute() at (4): ");
    } finally {                                                             //
      // Shut down the executor service:
      checkStates(es, "Before shutdown() at (6a): ");
      es.shutdown();                                                        //
      checkStates(es, "After shutdown() at (6a): ");

//    checkStates(es, "Before shutdownNow() at (6b): ");
//    es.shutdownNow();                                                     //
//    checkStates(es, "After shutdownNow() at (6b): ");
    }

    // Second phase of shutdown:
//  awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                           //
//  awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                      //
  }

  private static void checkStates(ExecutorService es, String msg) {        //
```

```
    System.out.printf("%-40s %-5s           %-5s%n",
        msg, es.isShutdown(), es.isTerminated());
  }

  private static void awaitAndShutdownNow(                                  //
      ExecutorService es, int timeout, TimeUnit timeunit) {
    try {
      // Timed wait for tasks to complete execution:
      if (!es.awaitTermination(timeout, timeunit)) {                        //
        // Attempt to cancel any uncompleted and waiting tasks:
        checkStates(es, "Before shutdownNow() at (11): " );
        es.shutdownNow();                                                   //
        checkStates(es, "After shutdownNow() at (11): " );
        // Timed wait for tasks to be cancelled:
        while (!es.awaitTermination(timeout, timeunit)) {                   //
          System.out.println("All tasks not yet completed at (12).");
        }
      }
      checkStates(es, "After awaitTermination() at (10): " );
    } catch (InterruptedException ie) {                                     //
      // Attempt to cancel any uncompleted and waiting tasks:
      es.shutdownNow();                                                     //
      // Reinstate the interrupt status.
      Thread.currentThread().interrupt();
      checkStates(es, "After interruption: ");
    }
  }
}
```

Probable output from the program when calling the `shutdown()` method at (6a):

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
Before shutdown() at (6a):               false           false

pool-1-thread-3 => dice value: 6
pool-1-thread-1 => dice value: 2
pool-1-thread-2 => dice value: 1
After shutdown() at (6a):                true            false
```

Probable output from the program when calling the `shutdownNow()` method at (6b):

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
pool-1-thread-3 => dice value: 4
pool-1-thread-2 => dice value: 5
pool-1-thread-1 => dice value: 5
Before shutdownNow() at (6b):            false           false
After shutdownNow() at (6b):             true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

Example 23.1 implements the necessary steps for using an executor service:

- *Create the executor service.*

  The utility class `Executors` provides methods for creating a wide range of executor services (Table 23.2). Appropriate executor service is created based on the desired task execution policy. In Example 23.1, a fixed thread pool with three worker threads is created at (2) to execute three tasks.

  Click here to view code image

  ```
  ExecutorService es = Executors.newFixedThreadPool(3);
  ```

- *Submit tasks to the executor service.*

  The task submitted to the executor service is defined as `Runnable` at (1). It simulates a dice roll, and when executed, it prints the dice value. The executing thread then sleeps for 100 milliseconds to prolong the execution time for the task to illustrate executor shutdown and termination.

  The tasks are submitted to the executor service using the `execute()` method— other methods are explored later (p. 1436, p. 1440). Each call to the `execute()` method will result in a new task being submitted. The implementation of the `Runnable diceRoll` will be executed three times, each time as a separate task.

  Click here to view code image

  ```
  es.execute(diceRoll);
  es.execute(diceRoll);
  es.execute(diceRoll);
  ```

- *Shut down and terminate the executor service.*

  When the executor service is no longer needed, it should be closed properly so that resources associated with the executor service can be reclaimed. As an executor service is not `AutoCloseable`, we cannot use the `try`-with-resources statement. The simplest shutdown procedure involves either calling the `shutdown()` or `shutdownNow()` method for this purpose. Calling either of these methods shuts down the executor service, after which any attempts to submit new tasks to the executor service will be rejected.

However, the `shutdown()` method will allow actively executing tasks and any waiting tasks to complete, but the `shutdown-Now()` method will attempt to cancel actively executing tasks and halt the processing of any waiting tasks.

The shutdown procedure is typically performed in a `finally` block associated with a `try` block containing the code that uses the executor service. This setup ensures that the executor service will always be shut down no matter how the tasks execute.

The method `checkStates()` at (9) in Example 23.1 is used to determine whether the executor service has been shut down and/or terminated at various stages in the execution, as can be seen in the program output.

The code below from Example 23.1 sketches the steps involved in using an executor:

Click here to view code image

```
// Create the executor service:
ExecutorService es = Executors.newFixedThreadPool(3);
try {
  // Submit tasks:
  es.execute(diceRoll);
  es.execute(diceRoll);
  es.execute(diceRoll);
} finally {
  // Shut down the executor service:
  es.shutdown();

//es.shutdownNow();
}
```

Two scenarios are shown in the output from the program in Example 23.1, corresponding to using the `shutdown()` and the `shutdownNow()` methods at (6a) and (6b), respectively. These methods do *not* wait for previously submitted tasks to complete execution before returning. Note that the `isTerminated()` method returns `false` after shutdown as one or more threads might not have completed execution—in this particular case, most probably waiting for sleep to time out. It is also important to note that the JVM will not terminate until all active threads in the thread pool have completed execution—analogous to normal threads in an application.

In the output, we see that the default name of a worker thread is composed of a thread pool number and a thread number—for example, `pool-1-thread-3`.

The output from the program when calling the `shutdown()` method at (6a) shows that the executor service did shut down, but the executor service did *not* terminate as some tasks continued execution—the method `isTerminated()` returned `false` when called after return from the `shutdown()` method. However, any running or submitted tasks were allowed to complete execution before the JVM terminated—in this particular case, threads were allowed to sleep until timed out.

In contrast, calling the `shutdownNow()` method cancels any executing and pending tasks. The `shutdownNow()` method does not wait for pending tasks to terminate.

From the output, we can see that calling the `isTerminated()` method on the executor returns `false`, as the executor has not terminated because all pending tasks have not been cancelled yet. The output from the program shows that, when cancelled, all three threads were interrupted while asleep. The `Interrupted-Exception` being caught and handled when the threads woke up and continued execution resulted in the tasks completing execution and the JVM being able to terminate.

Controlling Shutdown and Termination of an Executor Service

The `ExecutorService` interface provides the `awaitTermination()` method, which in conjunction with the shutdown methods, allows more refined control over pending tasks at shutdown.

Further control in shutting down and terminating an executor service is implemented by the `awaitAndShutdownNow()` method defined at (9) in Example 23.1. This method utilizes both the `awaitTermination()` method and the `shutdownNow()` method of the executor service. The executor service to shut down and the timeout to wait for pending tasks to complete are passed as parameters to the `awaitAndShutdownNow()` method.

The `awaitAndShutdownNow()` method first calls the `awaitTermination()` method at (10) that blocks until one of the following events occurs:

- The `awaitTermination()` method returns `true`, indicating that all tasks have completed, and thereby the executor service has been shut down and terminated.
- The `awaitTermination()` method returns `false`, indicating that there are still uncompleted tasks. The `shutdownNow()` method is called at (11) to cancel any uncompleted and waiting tasks. The `awaitTermination()` method is called again in a `while` loop at (12) to wait for any tasks still pending completion. This loop terminates when all tasks have completed; if the current thread is interrupted while awaiting task completion, the control goes to (13).
- The current thread is interrupted while waiting for the blocking call to the `awaitTermination()` method to return, in which case the `InterruptedException` thrown is caught by the `catch` block at (13). A further attempt is made in the `catch` block to cancel any pending tasks by calling the `shutdownNow()` method at (14).

The `awaitAndShutdownNow()` method at best makes an attempt at allowing pending tasks to complete after shutdown. It is called at (7a) and (7b) in Example 23.1 to illustrate shutting down an executor service and controlling any of its pending tasks. (For this discussion, (6b) is commented out in Example 23.1.)

The duration of the wait for pending tasks in the call to the `awaitTermination()` method impacts the fate of the pending tasks. The following call:

Click here to view code image

```
awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                              // (7a)
```

can result in the following output from the program, showing that two seconds of waiting was sufficient for all tasks to complete execution at (10), causing the executor service to terminate:

Click here to view code image

```
                                           isShutdown()   isTerminated()
Before execute() at (4):                      false           false
After execute() at (4):                       false           false
Before shutdown() at (6a):                    false           false
pool-1-thread-2 => dice value: 4
pool-1-thread-3 => dice value: 1
pool-1-thread-1 => dice value: 2
After shutdown() at (6a):                     true            false
After awaitTermination() at (10):             true            true
```

However, the following call:

Click here to view code image

```
awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                     // (7b)
```

can result in the following output from the program, showing that a one-millisecond wait was insufficient for all tasks to complete execution at (10), resulting in the `shutdownNow()` method at (11) being called. Any pending tasks are thus interrupted and the `InterruptedException` is handled by each task. The `awaitTermination()` method returns `true` when called in the `while` loop at (12), indicating that the interrupted tasks completed their execution:

Click here to view code image

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
Before shutdown() at (6a):               false           false
pool-1-thread-2 => dice value: 3
pool-1-thread-1 => dice value: 1
pool-1-thread-3 => dice value: 1
After shutdown() at (6a):                true            false
Before shutdownNow() at (11):            true            false
After shutdownNow() at (11):             true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
After awaitTermination() at (10):        true            true
```

Defining Tasks That Return a Result

Implementations of the `java.util.concurrent.Callable<V>` functional interface represent tasks that are designed to be executed by threads—analogous to implementations of the `java.lang.Runnable` functional interface. Whereas we can create and start a thread by passing a `Runnable` to a constructor of the `Thread` class, this is not possible with a `Callable<V>` implementation, as `Thread` constructors do not accept a `Callable<V>` object. In order to execute an implementation of the `Callable<V>` functional interface as a task in a thread, we can use an executor service—a topic which we explore thoroughly later (p. 1436).

In contrast to the `Runnable` interface, the `Callable<V>` interface is generic. As opposed to the void `run()` method of the `Runnable` interface, the `call()` method of the `Callable<V>` interface returns a result from the task it represents and can throw checked exceptions.

As it is a functional interface, the implementation of its sole method `call()` can be provided by a lambda expression.

An executor service provides bulk execution of tasks grouped in a collection using its methods `invokeAny()` and `invokeAll()` (p. 1439). Such bulk tasks can only be implemented as `Callable<V>` tasks, and not as `Runnable` tasks.

Typically, a `Runnable` is used for long-running tasks, such as a server, whereas a `Callable<V>` is a one-shot task that needs to compute and return a result.

The code at (1) implements a `Callable<Integer>` that simulates a dice using a `Random` object (§8.7, p. 482). The method `call()` is invoked directly on this implementation at (2) to roll the dice. Later we will see how we can obtain the value of a dice roll asynchronously in a thread using an executor service (p. 1438).

Click here to view code image

```
Random rng = new Random();
Callable<Integer> dice = () -> rng.nextInt(1,7);  // (1)
try {
  Integer diceValue = dice.call();      // (2)
  System.out.println(diceValue);        // Prints a value in the interval [1,6]
} catch (Exception ex) {
  ex.printStackTrace();
}
```

The following method is defined in the `Callable<V>` functional interface:

> `V call() throws Exception`
>
> Computes a result, or throws an exception if unable to do so.

Representing Task Results

A `Future<V>` object represents the result of a task that is asynchronously executed— that is, it represents the result of a task that is executed sometime in the future after its submission. For example, methods for submitting or scheduling tasks with an executor service return immediately. When a submitted task is executed sometime in the future, its result is returned in a `Future<V>` object that can be queried to obtain the result. A `Future<V>` object also provides methods to determine whether the task it represents completed normally or was cancelled (p. 1444).

The API of the `Future<V>` interface shown below provides references to examples that compute task results.

Click here to view code image

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
      throws InterruptedException, ExecutionException, TimeoutException
```

The first method blocks if necessary for the computation to complete, and then retrieves the result contained in this `Future<V>` object. In other words, the method does not return unless the task completes execution.

The second method blocks if necessary at most for the duration of the specified `timeout` in order for the computation to complete, and then retrieves the result contained in this `Future<V>` object, if one is available.

Calling these methods on a computation that has been cancelled, throws a `java.util.concurrent.CancellationException`. See Example 23.5, p. 1446.

Click here to view code image

```
boolean cancel(boolean mayInterruptIfRunning)
```

Attempts to cancel the execution of the task this `Future<V>` object represents.

If this task is already completed or cancelled, or if could not be cancelled, the method has no effect. Otherwise, if this task has not started when the method is called, this task is never run.

If the task has already started, then the `mayInterruptIfRunning` parameter determines whether the thread executing this task is interrupted in an attempt to stop the task. If the `mayInterruptIfRunning` parameter is `true`, the thread executing the task is interrupted; otherwise, the task is allowed to complete.

This method returns `false` if the task could not be cancelled, typically because it has already completed; otherwise, it returns `true`. However, the return value from this method does not necessarily indicate whether the task is now cancelled. For that purpose, the `isCancelled()` method should be used. See Example 23.5, p. 1446.

```
boolean isCancelled()
```

Returns `true` if this task was *cancelled* before it completed normally. See Example 23.5, p. 1446.

```
boolean isDone()
```

Returns `true` if this task *completed*. Note that completion implies either normal termination, or an exception, or cancellation.

Submitting Tasks and Handling Task Results

Tasks executed by executor services are represented by implementations of either the `Runnable` or the `Callable<V>` functional interface. The following selected methods of the `ExecutorService` interface can be used for submitting tasks to an executor service:

Click here to view code image

```
Future<?> submit(Runnable task)
<V> Future<V> submit(Runnable task, V result)
<V> Future<V> submit(Callable<V> task)
```

All three methods submit a task for execution to the executor service. The first two methods submit a `Runnable` task, but the last one submits a value-returning `Callable<V>` task.

All three methods return a `Future<V>` object that represents the task. On *successful* execution of the task, the `isDone()` method of the `Future<V>` object will return `true`. The value returned in the `Future<V>` object can be obtained by calling the `get()` method on the `Future<V>` object. The `Future` object will contain the value `null`, or the specified `result` in the call, or the result of executing the value-returning `Callable<V>` task, respectively, for the three methods. See Example 23.2, p. 1438.

Click here to view code image

```
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks)
                              throws InterruptedException
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks,
                              long timeout, TimeUnit unit)
                              throws InterruptedException
```

Executes the given tasks and returns a list of `Future<V>` holding their status and results when *all* complete, or if the timeout expires first as in the second method. The `isDone()` method on each `Future<V>` object in the returned list of `Future<V>` instances will return `true`.

Upon return from the second method, any uncompleted tasks are cancelled through interruption. A *completed task* is one that has terminated either normally or by throwing an exception. The order of the `Future<V>` instances returned is the same as the `Callable<V>` tasks in the collection passed as a parameter to the methods. See Example 23.2, p. 1438.

Click here to view code image

```
<V> V invokeAny(Collection<? extends Callable<V>> tasks)
                throws InterruptedException, ExecutionException
<V> V invokeAny(Collection<? extends Callable<V>> tasks,
                long timeout, TimeUnit unit)
                throws InterruptedException, ExecutionException,
                       TimeoutException
```

Executes the given tasks, returning the result of the first one that has *completed successfully without throwing an exception*, if any do succeed—or before the given timeout elapses in the case of the second method. Upon return, any uncompleted tasks are cancelled. See Example 23.2, p. 1438.

Submitting Tasks Individually

Example 23.2 illustrates submitting tasks *individually* and handling the task result. A `Runnable` task is defined at (1) that prints a dice value, and a `Callable<Integer>` task is defined at (2) that returns a dice value. We can of course call the `run()` and the `call()` methods to execute the `Runnable` `printDiceValue` and the `Callable<Integer>` `diceRoll`, respectively, but we would like to execute these tasks in a thread. For this purpose, a fixed thread pool is created at (3) in Example 23.2. Recall that this executor service reuses threads, and tasks may have to wait if all threads are occupied.

The overloaded `submit()` method of the `ExecutorService` interface can be used to submit tasks to the executor service. When a task submitted by these methods is executed asynchronously sometime in the future, the status or the result of executing the task is returned in a `Future<V>` object. The `get()` method of the `Future<V>` interface can be used to extract the value contained in a `Future<V>` object. However, note that the `get()` method will block the current thread until the task has completed. A timed version of the `get(timeout, timeunit)` method can also be used if there is a risk of waiting indefinitely.

The `Runnable` `printDiceValue` is submitted to the executor service at (4). The `get()` method, called at (5) to extract the value in the `Future<?>` object, blocks for the task to complete, if necessary. Since a `Runnable` does not return a value, the `get()` method returns the literal value `null` to indicate successful completion of the task. In this particular case, the type of the object returned is `Future<?>`, which is parameterized with the `?` wildcard to express that it contains an object of some unknown type.

The `Callable<Integer> diceValue` is submitted to the executor service at (6). The `Integer` result of executing this callable will be returned in a `Future<Integer>` object and can be extracted by calling the `get()` method on this object. The `get()` method blocks if the task has not completed execution.

Example 23.2 *Submitting Tasks and Handling Task Results*

Click here to view code image

```
package executors;
import java.util.List;
import java.util.concurrent.*;
import java.util.stream.Collectors;

public class TaskExecution {

  public static void main(String[] args) {
    // Runnable:
    Runnable printDiceValue = () ->                                         //
      System.out.println("Execution of Runnable: "
                        + ThreadLocalRandom.current().nextInt(1,7));

    // Callable<V>:
    Callable<Integer> diceRoll =                                            //
        () -> ThreadLocalRandom.current().nextInt(1,7);

    // Executor service:
    ExecutorService exs = Executors.newFixedThreadPool(3);                  //
    try {
      // Executing Runnable in a thread:
      Future<?> rfuture = exs.submit(printDiceValue);                       //
      Object result = rfuture.get();                                        //
      System.out.println("Result of Runnable: " + result);

      // Executing Callable<V> in a thread:
      Future<Integer> cfuture = exs.submit(diceRoll);                       //
      Integer diceValue = cfuture.get();                                    //
      System.out.println("Result of Callable: " + diceValue);

      // Executing bulk tasks:
      List<Callable<Integer>> callables                                     //
          = List.of(diceRoll, diceRoll, diceRoll);

      List<Future<Integer>> allFutures = exs.invokeAll(callables);          //
      List<Integer> resultList = allFutures.stream()                        //
          .map(future -> {
            try {
              return future.get();
            } catch(InterruptedException | ExecutionException ie) {
              throw new IllegalStateException(ie);
            }
```

```
                })
                .toList();
            System.out.println("Result of invokeAll(): " + resultList);

            Integer anyResult = exs.invokeAny(callables);                         //
            System.out.println("Result of invokeAny(): " + anyResult);
        } catch(InterruptedException | ExecutionException ie) {
            ie.printStackTrace();
        } finally {
            exs.shutdown();
        }
    }
}
```

Probable output from the program:

Click here to view code image

```
Execution of Runnable: 4
Result of Runnable: null
Result of Callable: 2
Result of invokeAll(): [1, 3, 6]
Result of invokeAny(): 3
```

Invoking Tasks Collectively

Example 23.2 also illustrates invoking tasks *collectively* and handling the task results. An executor service defines the blocking methods `invokeAll()` and `invokeAny()` that allow a collection of `Callable<V>` tasks to be executed—thus implementing *bulk task execution.*

A list of `Callable<Integer>` tasks is defined at (8) in Example 23.2, where each task is represented by the `Callable<Integer> diceRoll` defined at (2). The list represents tasks for rolling three dice. This list is executed by calling the `invokeAll()` method of the executor service at (9). In contrast to the `submit()` method, the `invokeAll()` method *blocks* until *all* tasks have completed, and their results are returned in a list of `Future<Integer>` objects. The order of the `Future<Integer>` objects in the result list corresponds to the order of the tasks in the task list. The resulting list of `Future<Integer>` objects is processed in a stream at (10) to extract the result (i.e., an `Integer`) from each `Future<Integer>` object using the `get()` method, and the results are collected in a list of `Integer` that represents the results of throwing three dice.

On the other hand, the list of `Callable<Integer>` tasks is executed at (11) using the `invokeAny()` blocking method, which returns the result of the *first* task that completes successfully without throwing an exception. On return, any tasks not completed are cancelled. Note that the `invokeAny()` method does *not* return the result in a

`Future<Integer>` object.

Only `Callable<V>` tasks can be executed in bulk, as `Runnable` tasks cannot be passed as a parameter to the invoke methods.

The `ScheduledExecutorService` Interface

A `ScheduledExecutorService` is an `ExecutorService` that schedules tasks to be executed after a given delay or to be executed periodically. Table 23.2 shows scheduled executor services provided by the factory methods of the `Executors` class.

The methods of the `ScheduledExecutorService` interface that schedule tasks return an object that implements the `ScheduledFuture<V>` interface. This interface extends the `Future<V>` interface, and thus provides methods to retrieve task results, to check execution status, or to cancel task execution.

Scheduling One-Shot Tasks

The overloaded `schedule()` method of the `ScheduledExecutorService` interface allows either a `Runnable` or a `Callable<V>` task to be scheduled for execution *only once after a specified delay*—that is, the `schedule()` methods execute *scheduled one-shot tasks*. The overloaded `schedule()` method of the `ScheduledExecutorService` interface is the analog of the overloaded `submit()` method of the `ExecutorService` interface for executing `Runnable` and `Callable<V>` tasks. These methods do not block—that is, they do not wait for the task to complete execution.

The following methods are defined in the `ScheduledExecutorService` interface that extends the `ExecutorService` interface:

> Click here to view code image
>
> ```
> ScheduledFuture<?> schedule(Runnable task, long delay, TimeUnit unit)
> ```
>
> Schedules a one-shot task that becomes enabled after the given delay. The method returns a `ScheduledFuture<?>` object representing pending completion of the task and whose `get()` method will return `null` upon completion.
>
> Click here to view code image
>
> ```
> <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay,
>                                 TimeUnit unit)
> ```

> Schedules a value-returning one-shot task that becomes enabled after the given delay. The method returns a `ScheduledFuture<V>` object which can be used to extract the result computed by the scheduled task or to cancel the scheduled task.

Example 23.3 illustrates scheduling one-shot `Runnable` and `Callable<V>` tasks for execution using a scheduled executor service. The example uses a *scheduled* thread pool with two threads that is created at (1) by calling the `newScheduledThreadPool()` factory method of the `Executors` class.

The `Runnable` task defined at (2) is passed to the `schedule()` method of the scheduled executor service at (3) to be executed once after a delay of 500 milliseconds. The `get()` method invoked on the resulting `ScheduledFuture<?>` object returns the value `null`, as expected, after execution of the `Runnable` task to indicate successful completion.

The `Callable<String>` task defined at (4) is passed to the `schedule()` method of the scheduled executor service at (5) to be executed once after a delay of one second. In this case, the `get()` method invoked on the resulting `ScheduledFuture<String>` object returns a `String` value after successful completion of the task.

Example 23.3 *Executing Scheduled One-Shot Tasks*

Click here to view code image

```
package executors;
import java.util.concurrent.*;

public class ScheduledOneShotTasks {
  public static void main(String[] args) {
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);   // (1
    try {
      // Schedule a one-shot Runnable:
      Runnable runnableTask                                               // (2
          = () -> System.out.println("I am a one-shot Runnable task.");
      ScheduledFuture<?> scheduledRunnableTaskFuture                      // (3
          = ses.schedule(runnableTask, 500, TimeUnit.MILLISECONDS);
      System.out.println("Runnable result: " + scheduledRunnableTaskFuture.get

      // Schedule a one-shot Callable<String>:
      Callable<String> callableTask
          = () -> "I am a one-shot Callable task.";                       // (4
      ScheduledFuture<String> scheduledCallableTaskFuture
          = ses.schedule(callableTask, 1, TimeUnit.SECONDS);              // (5
      System.out.println("Callable result: " + scheduledCallableTaskFuture.get
    } catch (InterruptedException | ExecutionException exc) {
      exc.printStackTrace();
    } finally {
        ses.shutdown();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
I am a one-shot Runnable task.
Runnable result: null
Callable result: I am a one-shot Callable task.
```

Scheduling Periodic Tasks

The `ScheduledExecutorService` interface provides two methods that can be used to schedule a `Runnable` task that should be executed periodically either after a fixed delay or at a fixed rate, with the first task execution commencing after an initial delay—in other words, scheduling *periodic tasks* for execution. No analogous methods are provided by the `ScheduledExecutorService` interface for `Callable<V>` tasks.

> Click here to view code image
>
> ```
> ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay,
>                                           long delay, TimeUnit unit)
> ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay,
>                                        long period, TimeUnit unit)
> ```
>
> The `scheduleWithFixedDelay()` method schedules a periodic task that becomes enabled first after the given initial delay, and subsequently with the given `delay` *between the termination of one execution and the commencement of the next*. This means that the time interval between the *start* of two consecutive task executions is the sum of the execution time of the first task execution and the specified delay between them.
>
> The `scheduleAtFixedRate()` method schedules a task that becomes enabled periodically for execution, with the first task execution commencing after the initial delay. This means that the time *between the start of two consecutive task executions is equal to the* `period`. Note that a new task execution commences after each period, regardless of whether or not the previous task execution has finished.
>
> For both methods, task executions continue indefinitely, unless one of the following events occurs: The task is cancelled via the returned future, or the executor shuts down, or an exception is thrown during task execution. The `isDone()`

> method on the returned future will then return `true`.

Example 23.4 illustrates scheduling `Runnable` tasks for repeated execution using a scheduled executor service. The `Runnable` task to be repeatedly executed is defined at (1). It prints information at the start and just before finishing its execution, which also includes the current time, by calling the auxiliary method `printTimestamp()` defined at (2). In between printing the information, the `Runnable` task sleeps for one second.

First scheduling a periodic task to be run with *a fixed delay* is illustrated in the `main()` method. An appropriate scheduled executor service is created at (3), namely, a scheduled thread pool of size 4. The method `scheduleWithFixedDelay()` is called at (4), passing the task, the initial delay of one second before commencement of the first task, and a fixed delay of three seconds between the end of one task execution and the start of the next one. The main thread sleeps for 15 seconds at (5) to allow the periodic task to be scheduled and executed. And finally the scheduled executor service is shut down at (6), which will allow the executing task to complete and the executor service to terminate. Without the shutdown, the periodic task would continue to be scheduled for execution. The output from the program shows that for the fixed delay, the delay between the finishing time of a task execution and the start time of the next one is approximately three seconds as it was specified—ignoring the nanoseconds in the current time.

Analogously, the scenario above is repeated for scheduling *a periodic task* to be executed at a fixed rate in the `main()` method. A new scheduled executor service is created at (7) for this purpose. This time the method `scheduleAtFixedRate()` is called at (8), passing the task, the initial delay of one second before commencement of the first task, and a fixed rate of three seconds between the *start* of each task execution.

The main thread sleeps for 10 seconds at (9) to allow the periodic task to be executed. The `shutdown()` method at (10) takes care of terminating the periodic task and the executor. The output from the program shows that for the fixed rate, the time between the start time of two consecutive task executions is approximately three seconds—again, ignoring the nanoseconds in the current time.

Example 23.4 *Executing Scheduled Periodic Tasks*

Click here to view code image

```
package executors;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.*;

public class ScheduledPeriodicTasks {

  private static Runnable task = () -> {                                  // (
    printTimestamp(" Start:  I am on it!");
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    printTimestamp(" Finish: I am not on it!");
  };

  private static void printTimestamp(String msg) {                        // (
    String threadName = Thread.currentThread().getName();
    String ts = LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm:ss")
    System.out.println(threadName + ": " + ts + msg);
  }

  public static void main(String[] args) {
    // Schedule a periodic task with fixed delay:
    ScheduledExecutorService ses1 = Executors.newScheduledThreadPool(4);  // (
    try {
      System.out.println("Fixed delay:");
      ses1.scheduleWithFixedDelay(task, 1, 3, TimeUnit.SECONDS);          // (
      TimeUnit.SECONDS.sleep(15);                                         // (
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      ses1.shutdown();                                                    // (
    }

    // Schedule a periodic task at fixed rate:
    ScheduledExecutorService ses2 = Executors.newScheduledThreadPool(4);  // (
    try {
      System.out.println("\nFixed rate:");
      ses2.scheduleAtFixedRate(task, 1, 3, TimeUnit.SECONDS);             // (
      TimeUnit.SECONDS.sleep(10);                                         // (
    } catch (InterruptedException e) {
      e.printStackTrace();

    } finally {
      ses2.shutdown();                                                    // (
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
Fixed delay:
pool-1-thread-1: 12:55:54 Start:  I am on it!
pool-1-thread-1: 12:55:55 Finish: I am not on it!
pool-1-thread-1: 12:55:58 Start:  I am on it!
pool-1-thread-1: 12:55:59 Finish: I am not on it!
pool-1-thread-2: 12:56:02 Start:  I am on it!
pool-1-thread-2: 12:56:03 Finish: I am not on it!
pool-1-thread-1: 12:56:06 Start:  I am on it!
pool-1-thread-1: 12:56:07 Finish: I am not on it!
```

```
Fixed rate:
pool-1-thread-2: 12:56:09 Start:  I am on it!
pool-1-thread-2: 12:56:10 Finish: I am not on it!
pool-1-thread-4: 12:56:12 Start:  I am on it!
pool-1-thread-4: 12:56:13 Finish: I am not on it!
pool-1-thread-4: 12:56:15 Start:  I am on it!
pool-1-thread-4: 12:56:16 Finish: I am not on it!
pool-1-thread-4: 12:56:18 Start:  I am on it!
pool-1-thread-4: 12:56:19 Finish: I am not on it!
```

Task Cancellation

Task cancellation is an important aspect of controlling task execution. Here we look at scenarios for cancelling tasks by invoking the `cancel()` method of the `Future<V>` object that represents the task after the task has been submitted to the executor. The effect of the `cancel()` method on the task depends on what stage the task is in when the `cancel()` method is called.

- *The task is in the queue, waiting to be assigned to a thread.*

  Calling the `cancel()` method on the `Future<V>` object that represents the task removes the task from the queue—thus cancelling the task.

- *The task is being executed by a thread.*

  Calling the `cancel()` method on the `Future<V>` object that represents the task sends an *interrupt signal* to the thread, the outcome of which depends on how the task handles the signal. In this section, we take a closer look at how cancellation can be organized programmatically when the thread that is executing a task is stopped by calling the `cancel()` method. See also the discussion on handling interrupt signals when a thread is interrupted by calling the `Thread.interrupt()` method (§22.4, p. 1393).

- *The task has completed execution.*

  Calling the `cancel()` method on the `Future<V>` object that represents the task has no effect on the task.

  Example 23.5 illustrates task cancellation by calling the `cancel()` method on the `Future<V>` object that represents the task.

  A computation-intensive task is defined by the `bigFactorial()` method at (1) to iteratively compute the factorial of a `BigInteger`. In each iteration of its `for(;;)` loop, the method checks at (2) whether the thread executing the method has been interrupted. If that is the case, it throws an `InterruptedException`. This terminates the `for(;;)` loop. The exception is caught and handled by the `catch` block at (3), causing the method to return normally. If there is no interrupt signal, the `for(;;)` loop continues the factorial computation. It is important that the task defines an appropriate action when the thread is interrupted.

The `timedTaskCancellation()` method at (4) illustrates using the timed `get()` method at (7) to wait for the task submitted at (6) to either finish execution and print the result at (8), or time out and throw a `TimeoutException` if the task did not complete execution. The output from the program shows that a `TimeoutException` was thrown. This exception is caught by the `catch` block at (9). The task is cancelled in the `catch` block by calling the `cancel()` method on the `Future<BigInteger>` object at (10). Since the `cancel()` method at (10) is called with the value `true`, the thread executing the task represented by the future is interrupted in an attempt to stop the task. The `cancel()` method returns `true`, indicating that the thread was interrupted. Whether the task is now cancelled is determined by the calling the `isCancelled()` method of the `Future<BigInteger>` object at (11) that returns the value `true`, verifying the cancellation. As only a single task is executed in the scenario above, we have used a single thread executor that is created at (5). The output from this scenario depends on the execution time to compute the factorial for the specified number at (6) and the duration of the timeout in the `get()` method at (7).

The `scheduledTaskCancellation()` method at (13) illustrates how to cancel a submitted task by scheduling a second task to do the cancellation, if necessary. As there are two tasks involved, a scheduled thread pool of size 2 is created at (14). The first task is submitted at (15) to compute the factorial of a big number. A second task is scheduled at (16) to call the `cancel()` method on the `Future<BigInteger>` object representing the first task, after a delay of 10 milliseconds. If the first task has completed before the `cancel()` method is called, it has no effect on the first task. However, if the first task has not completed execution before the second task executes, the `cancel()`

method will send an interrupt signal to the thread of the first task in an attempt to stop it. The output from the program shows that the first task was interrupted by the scheduled task. The `isCancelled()` method called on the `Future<BigInteger>` object at (18) confirms that the task was indeed cancelled. The output in this case depends on the execution time to compute the factorial for the specified number at (15) and the delay specified in the `schedule()` method at (16), before the call to the `cancel()` method is executed.

In summary, task cancellation is dependent on exploiting the thread interrupt system and providing an appropriate action in the interrupted thread executing the task.

Example 23.5 *Task Cancellation*

Click here to view code image

```
package executors;
import java.math.BigInteger;
import java.util.concurrent.*;

public class TaskCancellation {

  // Computation-intensive task:
  private static BigInteger bigFactorial(int number) {                    // (
    String threadName = Thread.currentThread().getName();
    BigInteger factorial = BigInteger.ONE;
    try {
      for (int i = 2; i <= number; i++) {
        factorial = factorial.multiply(BigInteger.valueOf(i));
        if (Thread.interrupted()) {                                       // (
          throw new InterruptedException();
        }
      }
      System.out.println(threadName + " completed.");
    } catch (InterruptedException e) {
      System.out.println(threadName + " interrupted.");                   // (
    }
    return factorial;
  }

  private static void timedTaskCancellation() {                           // (
    System.out.println("Timed task cancellation:");
    int number = 100000;
    long timeout = 1;
    TimeUnit unit = TimeUnit.SECONDS;
    System.out.println("NUMBER: " + number + ", "
                     + "TIMEOUT: " + timeout + " " + unit);
    ExecutorService exs = Executors.newSingleThreadExecutor();            // (
    Future<BigInteger> future = null;
    try {
      future = exs.submit(() -> bigFactorial(number));                    // (
      BigInteger result = future.get(timeout, unit);                      // (
      System.out.println("Factorial: " + result);                         // (
    } catch (TimeoutException e) {                                        // (
      System.out.println(e);
      System.out.println("cancel(true):  " + future.cancel(true));         // (
      System.out.println("isCancelled(): " + future.isCancelled());        // (
    } catch (InterruptedException | ExecutionException e) {
      System.out.println(e);                                              // (
    } finally {
      exs.shutdown();
    }
  }

  private static void scheduledTaskCancellation() {                       // (
    System.out.println("Scheduled task cancellation:");
    int number = 100000;
    long delay = 10;
    TimeUnit unit = TimeUnit.MILLISECONDS;
    System.out.println("NUMBER: " + number + ", DELAY: " + delay + " " + unit)
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);   // (
    try {
      Future<BigInteger> future = ses.submit(() -> bigFactorial(number)); // (
      ses.schedule(() -> future.cancel(true), delay, unit);               // (
      TimeUnit.SECONDS.sleep(1);                                          // (
      System.out.println("isCancelled(): " + future.isCancelled());        // (
    } catch (InterruptedException e) {
      System.out.println(e);
    } finally {
      ses.shutdown();
    }
  }

  public static void main(String[] args) {
    timedTaskCancellation();
    System.out.println();
    scheduledTaskCancellation();
  }
}
```

Probable output from the program:

Click here to view code image

```
Timed task cancellation:
NUMBER: 100000, TIMEOUT: 1 SECONDS
java.util.concurrent.TimeoutException
pool-1-thread-1 interrupted.
cancel(true):  true
isCancelled(): true

Scheduled task cancellation:
NUMBER: 100000, DELAY: 10 MILLISECONDS
pool-2-thread-1 interrupted.
isCancelled(): true
```

23.3 The Fork/Join Framework

To harness the benefits of multicore architectures, the Fork/Join Framework in the `java.util.concurrent` package provides support for parallel programming that allows computations to run in parallel on multiple processors. In this section we provide an introduction to this framework which is also the basis of parallel streams (§16.9, p. 1009), but it is also worth exploring further in its own right.

The Fork/Join Framework is specially suited for problems that can be solved by the *divide-and-conquer* problem-solving paradigm. This technique is prominent in processing data in large arrays and collections—for example, in designing sorting and searching algorithms. The main step in a divide-and-conquer algorithm is to divide the task repeatedly into subtasks of the same type until the task size is smaller than a given threshold, such that a subtask can be solved directly. A simplified divide-and-conquer algorithm to solve a problem is presented below. When the task can be solved directly, this is called the *base case*. Otherwise, the algorithm proceeds with dividing the task into subtasks, solving each subtask recursively, and combining the results from each subtask, all the time the idea being that each subtask gets smaller and moves toward the base case. Solving a subtask recursively is referred to as *forking*, and combing the results of subtasks is referred to as *joining*, hence the name Fork/Join Framework.

Click here to view code image

```
if (taskSize < threshold)
  solve task directly                          // Base case
else {
  divide task into subtasks                    // Divide
  solve each subtask recursively               // Conquer
  combine results from each subtask            // Combine
}
```

Each subtask is solved by invocation of the divide-and-conquer algorithm recursively. The subtasks are candidates for *parallel processing*. The Fork/Join Framework facilitates both the management of threads and the use of available processors for parallel processing of subtasks.

Abstract classes shown in Table 23.3 can be extended to define tasks. The abstract classes `RecursiveAction` and `RecursiveTask<V>` provide the abstract method `compute()` that defines the computation of the task. This method emulates the divide-and-conquer paradigm.

There are different ways to fork a subtask. One convenient way is to use the overloaded `invokeAll()` static method of the `ForkJoinTask<V>` class. This static method initiates the execution of the tasks and awaits their completion, returning any results, if necessary.

The overall execution of tasks is managed by the specialized executor service, `Fork-JoinPool`, shown in Figure 23.1. It is used analogous to the executors encountered in the Executor Framework. The initial task can be submitted to the fork-join pool by calling the `invoke()` method that commences the performing of the task.

Table 23.3 *Task-Defining Classes*

Classes in the java.util.concur	Description
`ForkJoinTask<V>`	Abstract class that defines a task
`RecursiveAction extends ForkJoinTask<Void>`	Abstract class that should be extended to define a task that does not return a value
`RecursiveTask<V> extends ForkJoinTask<V>`	Abstract class that should be extended to define a task that returns a value

Example 23.6 illustrates using the Fork/Join Framework. It counts the number of values in an array of random integers that satisfy a given predicate—in this particular case, all numbers divisible by 7. The `FilterTask` class at (1) extends the `RecursiveTask<Integer>` abstract class to define a task for this purpose. It implements the `compute()` method at (2) that mimics the divide-and-conquer algorithm.

The array to filter for numbers divisible by 7 is successively divided into two equal subarrays given by the indices `fromIndex`, `toIndex`, and `middle`. If the size of a subarray is less than the threshold, the base case at (3) is executed. It constitutes iterating over the subarray to do the counting, and returning the result. Note that the base case does not entail further subdivision of the task.

If the size of the array is over the threshold, two new filter tasks are created at (4) with two equal subarrays from the current subarray. The two new filter tasks are forked at (5) by calling the `invokeAll()` static method. Recursion occurs when the two subtasks are executed and their `compute()` method is called. The result returned from each subtask can be obtained by calling the inherited `join()` method from the superclass `ForkJoinTask`. These results are combined at (6) to return the combined result of the current subarray.

The `main()` method of the `ForkJoinDemo` class shows the steps to set up the fork-join pool for execution. The initial task is created at (9). The fork-join pool is create at (10) and invoked on the task at (11) to initiate the computation. The `invoke()` method returns the combined result of all the subtasks that were created in executing the initial task.

The powerful paradigm of parallel streams is evident from the stream-based solution at (12) when compared with the solution using the Fork/Join Framework explicitly.

Example 23.6 *Filtering Values Using Fork/Join Framework*

Click here to view code image

```
package forkjoin;

import java.util.Random;
import java.util.concurrent.*;
import java.util.function.IntPredicate;
import java.util.stream.IntStream;

class FilterTask extends RecursiveTask<Integer> {                         // (1
  public static final int THRESHOLD = 10_000;
  private int[] values;
  private int fromIndex;
  private int toIndex;
  private IntPredicate predicate;

  public FilterTask(int[] values, int fromIndex, int toIndex,
                    IntPredicate predicate) {
    this.values = values;
    this.fromIndex = fromIndex;
    this.toIndex = toIndex;
    this.predicate = predicate;
  }

  @Override
  protected Integer compute() {                                           // (2
    if (toIndex - fromIndex < THRESHOLD) {
      // The base case:                                                      (3
      var count = 0;
      for (int i = fromIndex; i < toIndex; i++) {
        if (predicate.test(values[i])) {
          ++count;
        }
      }
      return count;
    } else {
      // Create subtasks:                                                    (4
      var middel = fromIndex + (toIndex - fromIndex) / 2;
      var subtask1 = new FilterTask(values, fromIndex, middel, predicate);
      var subtask2 = new FilterTask(values, middel, toIndex, predicate);
      // Fork and execute the subtasks. Await completion.                    (5
      ForkJoinTask.invokeAll(subtask1, subtask2);
      // Combine the results returned by subtasks.                           (6
      return subtask1.join() + subtask2.join();
    }
  }
}

public class ForkJoinDemo {
  public static void main(String[] args) {
    // Set up the array with the random int values:                          (7
    final int SIZE = 1_000_000;
    int[] numbers = new Random().ints(SIZE).toArray();
    // Predicate to filter numbers divisible by 7:                          (8
    IntPredicate predicate = i -> i % 7 == 0;

    // Create the initial task.                                             (9
    var filterTask = new FilterTask(numbers, 0, numbers.length, predicate);
    // Create fork-join pool to manage execution of the task.               (1
    var pool = new ForkJoinPool();
    // Perform the task, await completion, and return the result:           (1
    var result = pool.invoke(filterTask);
    System.out.println("Fork/Join:       " + result);

    // Solution using parallel stream:                                      (1
    System.out.println("Parallel stream: " +
                        IntStream.range(0, numbers.length).parallel()
                                 .map(i -> numbers[i]).filter(predicate).count
  }
}
```

Probable output from the program:

```
Fork/Join:       142733
Parallel stream: 142733
```

23.4 Writing Thread-Safe Code

Thread-safety is a critical property of a concurrent application. Threads can generally execute their tasks concurrently without impending each other, unless they are sharing resources. Generally speaking, a shared resource can be any data that is visible to more than one thread. This could be an object or a primitive value, which multiple threads can access concurrently.

Each thread has its own execution stack that contains local variables of active methods during execution (§22.2, p. 1369). Any variable allocated on this stack is only visible to the thread that owns this stack, and is therefore automatically thread-safe. On the other hand, objects are placed in a heap which is shared by all threads. These objects are considered to be potentially visible to all threads. A thread can access any object in a heap if it has a reference to it. The challenge is to avoid thread interference (or race conditions) when concurrent threads are accessing a shared object.

As a running example, we will use different implementations of a counter to illustrate various approaches to achieving thread-safety. Example 23.7 shows the interface `ICounter` at (1) that is implemented by a counter. The interface defines the two methods `increment()` and `getValue()` to increment and read the value in a counter, respectively.

The class `TestCounters` at (3) tests various counter implementations from (5) to (10) in the `main()` method at (4) by calling the method `runIncrementor()` defined at (11). This method creates a `Runnable` incrementor at (12) that calls the `increment()` method of a counter a fixed number of times (`NUM_OF_INCREMENTS`). The incrementor is submitted a fixed number of times to an executor service in the `try` block at (13), corresponding to the size of the thread pool of the executor service (`POOL_SIZE`). All together, a counter is incremented 10,000 times (`NUM_OF_INCREMENTS*POOL_SIZE`).

The output shown in Example 23.7 is for the various counter implementations that will be discussed in this section. First up is an unsafe counter implementation at (2) which is not thread-safe, as we can see from the incorrect result in the output. The cause can be attributed to thread interference (§22.4, p. 1388) and memory consistency errors (§22.5, p. 1414) when accessing the shared counter field by the concurrent threads. We look at different solutions (in addition to the `synchronized` code) to implement mutually exclusive access of a shared object in a concurrent application.

Example 23.7 *Testing Counter Implementations*

Click here to view code image

```
package safe;
/** Interface that defines a basic counter. */
interface ICounter {                                                       // (
  void increment();
  int getValue();
}
```

Click here to view code image

```
package safe;
public class UnsafeCounter implements ICounter {                           // (
  private int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

Click here to view code image

```
package safe;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class TestCounters {                                                // (
  private static final int NUM_OF_INCREMENTS = 1000;
  private static final int POOL_SIZE = 10;

  public static void main(String[] args) throws InterruptedException {    // (

    UnsafeCounter usc = new UnsafeCounter();                              // (
    runIncrementor(usc);
    System.out.printf("Unsafe Counter: %24d%n", usc.getValue());

    VolatileCounter vc = new VolatileCounter();                           // (
    runIncrementor(vc);
    System.out.printf("Volatile Counter: %22d%n", vc.getValue());

    SynchronizedCounter sc = new SynchronizedCounter();                   // (
    runIncrementor(sc);
    System.out.printf("Synchronized Counter: %18d%n", sc.getValue());

    AtomicCounter ac = new AtomicCounter();                               // (
    runIncrementor(ac);
    System.out.printf("Atomic Counter: %24d%n", ac.getValue());

    ReentrantLockCounter rlc = new ReentrantLockCounter();                // (
    runIncrementor(rlc);
    System.out.printf("Reentrant Lock Counter: %16d%n", rlc.getValue());

    ReentrantRWLockCounter rwlc = new ReentrantRWLockCounter();           // (
    runIncrementor(rwlc);
    System.out.printf("Reentrant Read-Write Lock Counter: %d%n", rwlc.getValue
  }

  public static void runIncrementor(ICounter counter) {                   // (
    // A Runnable incrementor to call the increment() method of the counter
    // a fixed number of times:
    Runnable incrementor = () -> {                                        // (
      IntStream.rangeClosed(1, NUM_OF_INCREMENTS).forEach(i->counter.increment
    };

    // An executor service to manage a fixed number of incrementors:
    ExecutorService execService = Executors.newFixedThreadPool(POOL_SIZE);

    // Submit the incrementor to the executor service. Each thread executes
    // the same incrementor, and thereby increments the same counter.

    try {                                                                 // (
      IntStream.range(0, POOL_SIZE).forEach(i -> execService.submit(incremento
    } finally {
      execService.shutdown();
    }
    // Wait for all tasks to complete.
    try {
      while (!execService.awaitTermination(1, TimeUnit.SECONDS));
    } catch (InterruptedException e) {
      execService.shutdownNow();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
Unsafe Counter:                         8495
Volatile Counter:                       8765
Synchronized Counter:                  10000
```

```
Atomic Counter:                     10000
Reentrant Lock Counter:             10000
Reentrant Read-Write Lock Counter: 10000
```

Immutability

We have seen how to implement thread-safe access to shared resources in `synchronized` code that uses intrinsic locks (§22.4, p. 1387). However, if the shared resource is an immutable object, thread-safety comes for free.

An object is *immutable* if its state cannot be changed once it has been constructed. Since its state can only be read, there can be no thread interference and the state is always consistent.

Examples of immutable classes in the Java SE Platform API and designing immutable classes are discussed in §6.7, p. 356.

Volatile Fields

To optimize the execution of compiled code, the compiler performs optimizations on the bytecode. Typical optimizations are instruction reordering, method call inlining, and loop optimizations. For multithreaded applications, thread-specific optimizations can also be performed. It is quite common for threads to create their own private cached copies of variables (called *thread-local* variables) for performance reasons. Values of thread-local variables are synchronized with the values of their master variables in main memory at various times—for example, when a thread enters or exits synchronized code. Declaring a field as `volatile` informs the compiler that the field will be modified by different threads, and all reads and writes to the field should be reflected in the master copy in main memory. Without the `volatile` modifier, there is no guarantee that the latest value in a shared field will be visible to the different threads. The *visibility* of a shared `volatile` field is guaranteed by the following rule:

- *Volatile Field Rule:* A write to a `volatile` field *happens-before* every subsequent read of that field (§22.5, p. 1414).

The rule implies that a read on a `volatile` field will always see the value from the latest write on the `volatile` field. Without the `volatile` modifier, the threads may see different values in the shared field. The class `NonVolatileDemo` in Example 23.8 illustrates the potential problem that can occur with visibility when different threads share data. The `boolean` field `stopThread` at (1) is used in the `run()` method at (3) of each child thread spawned in the loop at (2) to stop the thread when the field value becomes `true`. The main thread sets this value after sleeping for a little while. If the threads are using thread-local copies of the field `stopThread`, there is no guarantee when these threads will get to see the updated value, and therefore, might continue executing forever.

The class `VolatileDemo` in Example 23.8 declares the field `stopThread` at (6) to be `volatile`. This is a typical use of the `volatile` modifier: to flag a condition to different threads. When the value is updated in the main thread, the child threads will see the updated value on the next read of the field, and are guaranteed to terminate.

Example 23.8 *Visibility of Shared Data*

Click here to view code image

```
package safe;
/** Potential problem with visibility of shared data. */
public class NonVolatileDemo {

  private static boolean stopThread = false;             // (1)

  public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 2; i++) {                        // (2)
      new Thread(() -> {
        while (!stopThread) {                            // (3)
          System.out.println(Thread.currentThread().getName()
                             + ": Get me out of here!");
        }
      }, "T" + i).start();
    }
    Thread.sleep(1);                                     // (4)
    stopThread = true;                                   // (5)
  }
}
```

Click here to view code image

```
package safe;
/** Volatile field to guarantee visibility of shared data. */
public class VolatileDemo {

  private static volatile boolean stopThread = false;    // (6)

  // The main() method remains the same as in the NonVolatileDemo class.

}
```

Probable output from each of the programs:

```
...
T0: Get me out of here!
T0: Get me out of here!
T0: Get me out of here!
T1: Get me out of here!
```

```
...
```

All *reads* and *writes* are *atomic actions* for all `volatile` fields, regardless of whether they are object references or are of primitive data types. An *atomic action* is guaranteed to be performed by a thread without any interleaving—that is, without any race conditions. Such an action either performs in its entirety or not at all—akin to a database transaction. Most program actions are not actually atomic; even something as simple as an arithmetic operator takes more than one CPU cycle, and therefore can be interrupted before the action completes.

Note that atomicity holds only for read and write operations, not for non-atomic operations like the increment operator (`++`), that are actually executed in several steps. For example, the expression `++i` is equivalent to the following code:

```
int tmp = i;
tmp = tmp + 1;
i = tmp;
```

The expression `++i` actually is being evaluated as a read of the value in `i`, and then a write to `i` after the value has been incremented by `1`. If `i` was a shared `volatile` field, a different thread could read it between the atomic read and write operations, resulting in potential memory consistency errors, unless intermediate values in `i` did not matter. Computing the next value in `i` is dependent on the previous value in `i`. Declaring a variable with such data dependencies as `volatile` may not be adequate, as demonstrated by Example 23.9 that implements the counter as a `volatile` field. Not surprisingly, the output from the program shows an incorrect result of incrementing the counter due to memory consistency errors.

The `volatile` keyword is enough to combat memory consistency problems. That is, it solves the visibility problem (writing to main memory), but not interleaving of operations on a shared variable (i.e., thread interference/race conditions). Solving both problems requires mutual exclusion. We explore solutions for implementing mutual exclusion other than `synchronized` code later in this section: atomic variables for mutual exclusion on shared single variables (p. 1456) and programmatic locks for mutual exclusion on shared resources (p. 1460).

Example 23.9 *Volatile Counter*

Click here to view code image

```
package safe;
public class VolatileCounter implements ICounter {
  private volatile int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

Probable output from the program in Example 23.7, p. 1451:

Click here to view code image

```
Volatile Counter:                    87672
```

There is no need to use a `volatile` field unless it is shared between threads, or if only atomic reads and writes are necessary on a field. A field of primitive type `long` or `double` does not guarantee atomic reads and writes, unless it is declared `volatile`. Also, a `final` field cannot be declared `volatile`, as it is immutable.

It is worth noting the differences between the `volatile` and the `synchronized` keywords. The keyword `volatile` is only applicable to fields, whereas the keyword `synchronized` is only applicable to a statement block or method. A `synchronized` statement cannot synchronize on `null`, but a `volatile` field may be `null`.

Since there are no locks involved when accessing a `volatile` field, there is no blocking of threads either. Performance overhead is also lightweight, compared to `synchronized` code which is bound by the whole regime of thread management, although the keyword signals to the compiler *not* to undertake certain optimizations. And while the `volatile` keyword cannot replace `synchronized` code in all situations, it is more efficient in certain situations where the visibility and atomicity of a shared field are overriding factors.

Atomic Variables

Unless the action on a shared `volatile` variable is atomic, thread-safety cannot be guaranteed because of potential race conditions. Non-atomic actions (such as the use of the increment operator) by different threads can result in interleaving of constituent actions like read, update, and write.

Shared lock-free thread-safe variables are implemented by the classes in the `java.util.concurrent.atomic` package (Table 23.4). The classes provide methods that implement *atomic actions*. The classes and the variables that denote their instances are called *atomic classes* and *atomic variables*, respectively.

Atomic variables also behave as if they are volatile, so there is no need to declare them explicitly with the `volatile` keyword—that is, the atomic actions of the atomic classes guarantee visibility of an atomic variable to other threads.

The atomic classes thus implement *non-blocking lock-free synchronization* of atomic variables by a predefined set of atomic actions, ensuring visibility and avoiding race conditions.

Table 23.4 *Selected Atomic Classes*

Atomic classes in the java.	**Mutable values that may be updated atomically**
`AtomicBoolean`	A `boolean` value
`AtomicInteger implements Number`	An `int` value
`AtomicLong implements Number`	A `long` value
`AtomicReference<V>`	An object reference

Example 23.11 illustrates a thread-safe counter implemented using the `Atomic-Integer` class. The counter field is declared at (1) as an `AtomicInteger` and initialized to `0`. The `getValue()` and `increment()` methods at (2) and (3) delegate their operation to appropriate atomic methods of the `AtomicInteger` class. As the name implies, the `get()` method atomically returns the current value of the counter. The `increment-AndGet()` method atomically increments the counter and returns the new value (that is ignored in this example). The `incrementAndGet()` method entails more than one action, but none of them can be interrupted. The output shows that the counter was correctly incremented by the threads.

To understand how some of the atomic operations are implemented, we take a look at the `while` loop at (5), which implements the equivalent of the `incrementAndGet()` method at (4).

The strategy used in the `while` loop is known as *CAS: Compare and Set*, after the method called at (8). The `compareAndSet()` method will not set the counter to the new value unless the current value is equal to the expected value.

Assume that the current value in the counter is 100. Referring to the numbered lines in the code:

(6) The expected value is 100 (i.e., the current counter value).

(7) The new value is 101, after incrementing the expected value.

(8) The call to the `compareAndSet()` method in the conditional expression of the `if` statement returns `true`, since the current value (100) is equal to the expected value (100). The new value (101) is set in the counter—that is, it is correctly incremented by 1 from the previous value.

(9) The loop terminates.

Assume that by the time the current value is read again for comparison at (8) by the `compareAndSet()` method, it has been changed to 105 by another thread. This is possible by another thread in the time between the counter value is accessed at (6) by the `get()` method and then again at (8) by the `compareAndSet()` method—that is, there is a potential race condition.

(8) The call to the `compareAndSet()` method in the conditional expression of the `if` statement returns `false`, since the current value (105) is not equal to the expected value (100). The value in the counter is not changed as it could corrupt the state of the counter, and the loop continues until the counter value is correctly incremented by 1.

The `while` loop ensures that the value is always incremented by 1 from the previous value. Many of the methods in the atomic classes use a similar strategy to implement atomic operations.

It goes without saying that the `AtomicInteger` class should not be used as a replacement for the `Integer` class. We leave the inquisitive reader to explore the Atomic API further at leisure.

Example 23.10 *Atomic Counter*

Click here to view code image

```
package safe;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter implements ICounter {
  private AtomicInteger counter = new AtomicInteger(0);                    // (

  @Override
  public int getValue() {                                                  // (
    return counter.get();
  }

  @Override
  public void increment() {                                                // (
    counter.incrementAndGet();                                             // (
//  while (true) {                                                         // (
//    int expectedValue = counter.get();                                   // (
//    int newValue = expectedValue + 1;                              // (
//    if (counter.compareAndSet(expectedValue, newValue)) {// (8) Compare and
//      return;                                               // (9)
//    }
//  }
  }
}
```

Output from the program in Example 23.7, p. 1451:

Click here to view code image

```
Atomic Counter:                   10000
```

The `AtomicInteger` class provides the following constructors:

> Click here to view code image
>
> ```
> AtomicInteger()
> AtomicInteger(int initialValue)
> ```
>
> Create a new `AtomicInteger` with initial value 0 or with the given initial value, respectively.

Selected atomic methods from the `AtomicInteger` class are presented below.

> ```
> int get()
> ```
>
> Returns the current value.
>
> ```
> void set(int newValue)
> ```
>
> Sets the value to `newValue`.
>
> ```
> int getAndSet(int newValue)
> ```
>
> Atomically sets the value to `newValue` and returns the old value.
>
> Click here to view code image
>
> ```
> boolean compareAndSet(int expectedValue, int newValue)
> ```
>
> Atomically sets the value to `newValue` if the current value == `expectedValue`, returning `true` if the operation was successful; otherwise, it returns `false`.
>
> ```
> int addAndGet(int delta)
> int getAndAdd(int delta)
> ```
>
> Atomically adds the given value to the current value, returning the updated value or the previous value, respectively.

```
int decrementAndGet()
int getAndDecrement()
```

Atomically decrements the current value, returning the updated value or the previous value, respectively.

```
int incrementAndGet()
int getAndIncrement()
```

Atomically increments the current value, returning the updated value or the previous value, respectively.

Click here to view code image

```
int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)
```

Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value or the previous value, respectively.

Click here to view code image

```
int updateAndGet(IntUnaryOperator updateFunction)
int getAndUpdate(IntUnaryOperator updateFunction)
```

Atomically updates the current value with the results of applying the given function, returning the updated value or the previous value, respectively.

```
int    intValue()
double doubleValue()
float  floatValue()
long   longValue()
```

Returns the current value of this `AtomicInteger` as an `int`, `double`, `float`, and `long` after a widening primitive conversion, if necessary, respectively.

Intrinsic Locking Revisited

Intrinsic locking provides a blocking, lock-based, thread-safe solution for concurrent threads accessing mutable shared resources. The keyword `synchronized` is used to implement critical sections of code (synchronized methods/blocks) that rely on intrinsic locks to guarantee mutual exclusion to a single thread at a time (§22.4, p. 1387). Race conditions are thus avoided by mutual exclusion guaranteed by the critical section. The happens-before object lock rule (in this case for intrinsic locks) guarantees that any updates to the shared data done by one thread in a critical section will be visible to any other thread in another critical section guarded by the same intrinsic lock—thus eliminating memory consistency errors (§22.5, p. 1414).

Example 23.11 is an implementation of a thread-safe counter using `synchronized` methods for this particular case. Note that it is not necessary to declare the `counter` field as `volatile`, as both visibility and absence of race conditions are guaranteed by the synchronized code. The two operations on the counter are implemented as `synchronized` methods that are mutually exclusive, ensuring thread-safe use of the counter when accessed by different threads.

Example 23.11 *Synchronized Counter*

Click here to view code image

```
package safe;
public class SynchronizedCounter implements ICounter  {
  private int counter = 0;

  @Override public synchronized int getValue() { return counter; }
  @Override public synchronized void increment() { counter++; }
}
```

Output from the program in Example 23.7, p. 1451:

Click here to view code image

```
Synchronized Counter:                10000
```

Programmatic Locking

The Lock API in the `java.util.concurrent.locks` package provides flexible programmatic locking mechanisms that can be used to ensure thread-safety of shared mutable resources accessed by concurrent threads.

Intrinsic locking is based on the *acquire-release lock paradigm* to implement mutual exclusion. This paradigm is also the basis of locking mechanisms for mutual exclusion provided by the Lock API. This API also allows more control over *lock acquisition* (i.e., how the lock should be acquired) and *lock disciplines* (i.e., how to choose a thread among the waiting threads to acquire a released lock). Programmatic locking is thus more flexible, and can prevent threads from potential resource starvation (§22.5, p. 1412).

Selected interfaces and classes from the Lock API are shown in Figure 23.3. The main interfaces are `Lock` and `ReadWriteLock` that are implemented by the `ReentrantLock` and the `ReentrantReadWriteLock` classes, respectively. The `ReentrantReadWriteLock` class defines two nested classes, `ReadLock` and `WriteLock`, that implement the `Lock` interface. We explore the Lock API in the next section.

Figure 23.3 *Selected Interfaces and Classes in the Lock API*

Reentrant Lock

The `ReentrantLock` class implements the `Lock` interface. Instances of the `Reentrant-Lock` class are basically analogous to intrinsic locks. A thread needs to *lock* a `ReentrantLock` instance in order to gain exclusive access to a critical section guarded by the `ReentrantLock` instance. Afterward, the `ReentrantLock` instance must be *unlocked* so that other threads can try to gain access to the critical region.

The `ReentrantLock` class provides two constructors to create locks, with or without a *fairness policy*, as shown below at (1) and (2), respectively. The fairness policy is specified by the value `true` in the constructor. In this case, the *acquisition order* is that the longest-waiting thread for the lock is chosen to acquire the released lock. Otherwise, no guarantees are given as to how a thread will be chosen from the threads waiting to acquire the lock.

Click here to view code image

```
Lock frl = new ReentrantLock(true);  // (1) Fair reentrant lock
Lock rl = new ReentrantLock();       // (2) Reentrant lock with no fairness po
```

> `ReentrantLock()`
>
> Creates an instance of `ReentrantLock`. This `ReentrantLock` instance does *not* use a fairness policy, and it is equivalent to `ReentrantLock(false)`.
>
> `ReentrantLock(boolean fair)`
>
> Creates an instance of `ReentrantLock` with the given fairness policy. The fair ordering policy is implied by the value `true`, allowing the longest-waiting thread to acquire the lock when it is released.

The basic idiom to use a reentrant lock for mutual exclusion is shown below. A thread first *locks* the `ReentrantLock` instance at (1) by calling the `lock()` method—we say that the thread *acquires* the lock. At (2), the thread *unlocks* the `ReentrantLock` instance by calling the `unlock()` method—we say that the thread *releases* the lock. The thread has exclusive access to the code executed between (1) and (2) if it acquires the lock. Using the `try-finally` statement ensures that the lock is always released in the `finally` block. In general, the `lock()` and `unlock()` methods can be called in separate methods.

Click here to view code image

```
frl.lock();                // (1) Acquire the lock.
try {
  // Exclusive access...
} finally {
  frl.unlock();            // (2) Release the lock.
}
```

The `ReentrantLock` class implements methods of the `Lock` interface that define different *lock acquisition policies*.

- *Unconditional locking*

  The `lock()` method implements this policy, where the thread must unconditionally wait to acquire the lock if it is not available immediately.

  Click here to view code image

  ```
  frl.lock();                                              // Unconditio
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
  ```

- *Polled locking*

  The `tryLock()` method implements this policy, where the call will return immediately with the value `false` if the lock is not available. The value `true` is returned if the lock is acquired.

  Click here to view code image

  ```
  if (frl.tryLock()) {                                      // Polled lo
    try { /* Lock acquired. Can access the resource. */ }
    finally { frl.unlock(); }
  } else { /* Lock was not acquired. */ }
  ```

- *Timed locking*

  The `tryLock(timeout, timeunit)` method will also return `false` if the thread is not able to acquire the lock in the specified time. If the thread is interrupted while waiting for the lock, it will throw an `InterruptedException`.

Click here to view code image

```
try {
  if (frl.tryLock(100, TimeUnit.MILLISECONDS)) {            // Timed lo
    try { /* Lock acquired. Can access the resource. */ }
    finally { frl.unlock(); }
  } else { /* Lock was not acquired. */ }
} catch (InterruptedException iex) { iex.printStackTrace(); }
```

- *Interruptible locking*

  The `lockInterruptibly()` method throws an `InterruptedException` if the thread is interrupted while acquiring the lock or while waiting for the lock.

  Click here to view code image

```
try {
  frl.lockInterruptibly();                                   // Interrupt
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
} catch (InterruptedException iex) { iex.printStackTrace(); }
```

More details are provided below about the lock acquisition policy implemented by the methods mentioned above. These methods are defined by the `Lock` interface and implemented by the `ReentrantLock` class.

> `void lock()`
>
> If the lock is not held by another thread, the *lock hold count* is set to 1 and the method returns immediately.
> If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns immediately.
> If the lock is held by another thread, the thread waits until the lock is acquired, at which point the lock hold count is set to 1.
>
> `void unlock()`
>
> If the current thread is the holder of this lock, the lock hold count is decremented. If the lock hold count now has the value `0`, the lock is released.
> If the current thread is not the holder of this lock, then an `IllegalMonitorState-Exception` is thrown.
> Note that a lock action on a lock must be matched by an unlock action.
> The thread must make sure that an acquired lock is released after use, typically in the `finally` block of a `try-finally` statement.

> ```
> boolean tryLock()
> ```
>
> If the lock is not held by another thread, the *lock hold count* is set to 1 and the method returns `true`. This method does not honor fairness—it does not care if there are any threads already waiting for this lock.
> If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns `true`.
> If the lock is held by another thread, the method returns immediately with the value `false`.
>
> Click here to view code image
>
> ```
> boolean tryLock(long time, TimeUnit unit) throws InterruptedException
> ```
>
> Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted. In this case, the lock hold count is set to 1 and the value `true` is returned. The method respects the fair ordering policy if one has been specified for this lock.
> If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns `true`.
> If the current thread is interrupted while waiting to acquire the lock, an `InterruptedException` is thrown.
> If timed out, the method returns the value `false`.

> Click here to view code image
>
> ```
> void lockInterruptibly() throws InterruptedException
> ```
>
> If the lock is not held by another thread and the current thread is not interrupted, the *lock hold count* is set to 1 and the method returns immediately.
> If the current thread already holds the lock, the lock hold count is incremented by 1 and the method returns immediately.
> If the lock is held by another thread, the current thread waits for the lock. If the lock is acquired by the current thread while waiting, the lock hold count is set to 1.
> If the current thread is interrupted while waiting to acquire the lock, an `InterruptedException` is thrown.

The following selected methods for querying a lock are only defined by the `ReentrantLock` class:

> ```
> boolean isFair()
> ```

> Returns `true` if this lock has the fairness policy set to `true`.
>
> ```
> boolean isLocked()
> ```
>
> If the lock is held by any thread, this method returns `true`; otherwise, it returns `false`.
>
> ```
> int getQueueLength()
> ```
>
> Returns an estimate of the number of threads waiting to acquire this lock.
>
> ```
> boolean hasQueuedThreads()
> ```
>
> Returns `true` if any threads are waiting to acquire this lock; otherwise, it returns `false`.
>
> Click here to view code image
>
> ```
> boolean hasQueuedThread(Thread thread)
> ```
>
> Returns `true` if the given thread is waiting to acquire this lock; otherwise, it returns `false`.

Example 23.12 illustrates using a reentrant lock to implement a thread-safe counter. The class `ReentrantLockCounter` at (1) implements the `ICounter` interface (Example 23.7). It defines a counter whose value can be read by the method `get-Value()` at (1), and incremented by the method `increment()` at (5). The class instantiates a `ReentrantLock` instance at (2) that is used in the two methods to implement exclusive access to the `counter` field.

The `getValue()` method at (4) acquires the `rl` lock, reads and stores the current `counter` value locally, unlocks the `rl` lock, and returns the locally stored `counter` value. The `increment()` method at (5) acquires the `rl` lock, increments the `counter`, calls the `getValue()` method (ignoring the returned value), and releases the `rl` lock. The setup with the locks ensures that any read or write operation will have exclusive access to the `counter` field. When testing the `ReentrantLockCounter` class with Example 23.7, p. 1451, the output shows that the counter was incremented correctly by the threads.

Note that the expression of a `return` statement in a `try` block is evaluated first and its value stored locally, before the `finally` block is executed. The stored result is returned after the completion of the `finally` block.

Intrinsic locks are *reentrant*: A thread that has acquired an intrinsic lock can acquire the same intrinsic lock again immediately (§22.4,

p. 1390). This is also true of the locks implemented by the `ReentrantLock` class. In other words, explicit locks implemented by the `ReentrantLock` class are reentrant, as their name implies. The following code relies on the lock being reentrant:

Click here to view code image

```
frl.lock();
frl.lock();
// Access the resource. Lock hold count is 2.
frl.unlock();
frl.unlock();
```

Example 23.12 also illustrates the reentrant nature of explicit locks. The `increment()` method of the `ReentrantLockCounter` class calls the method `getValue()`. The current thread already holds the `rl` lock when the `getValue()` method is called. The `get-Value()` method also calls the `lock()` method on the `rl` lock. If `rl` was not reentrant, the current thread would not be able to proceed in the `getValue()` method and would starve waiting for the lock to become available, as it already has been acquired by the current thread. The reentrant nature of the `rl` lock allows the current thread to reacquire the lock and execute the `getValue()` method.

Example 23.12 *Reentrant Lock Counter*

Click here to view code image

```
package safe;
import java.util.concurrent.locks.*;

public class ReentrantLockCounter implements ICounter {                 // (1)

  private Lock rl = new ReentrantLock();                               // (2)
  private int counter = 0;                                             // (3)

  @Override
  public int getValue() {                                              // (4)
    rl.lock();
    try {
      return counter;
    } finally { rl.unlock(); }
  }

  @Override
  public void increment() {                                            // (5)
    rl.lock();
    try {
      counter++;                                                       // (6)
      getValue();                                                      // (7)
    } finally { rl.unlock(); }
  }
}
```

Output from the program in Example 23.7, p. 1451:

Click here to view code image

```
Reentrant Lock Counter:          10000
```

Reentrant Read-Write Lock

The explicit lock implemented by the `ReentrantLock` class is *mutually exclusive*—that is, only one thread at a time can obtain the lock and thereby access the shared data. There are cases where shared data is read more often than it is modified—that is, there are more *readers* (or *reader threads*) than *writers* (or *writer threads*) accessing the shared data. Taking into consideration such behavior to improve the level of concurrency would be a challenge using a mutually exclusive lock, as the lock acquisition is exclusive, irrespective of whether it is a read or a write operation.

The `ReadWriteLock` interface provides a more flexible and sophisticated locking mechanism than a mutually exclusive lock for accessing shared data. Figure 23.3 shows that the `ReentrantReadWriteLock` class implements the `ReadWriteLock` interface. It implements a *read-write lock* that actually maintains *a pair of associated locks*, one for read operations and one for write operations. The *read lock* is an instance of the `ReadLock` inner class and the *write lock* is an instance of the `WriteLock` inner class, both of which implement the `Lock` interface. The `ReadWriteLock` interface defines the methods to obtain the read and the write lock of a read-write lock.

The following methods are defined in the `ReadWriteLock` interface:

```
Lock readLock()
Lock writeLock()
```

These methods are implemented by inner classes in the `ReentrantReadWriteLock` class:

> Click here to view code image
>
> ```
> ReentrantReadWriteLock.ReadLock readLock()
> ReentrantReadWriteLock.WriteLock writeLock()
> ```
>
> Return the lock used for reading and for writing, respectively. The inner classes `ReadLock` and `WriteLock` implement the `Lock` interface.

The `ReentrantReadWriteLock` class provides the following constructors to create read-write locks with or without a fairness policy:

> `ReentrantReadWriteLock()`
>
> Creates an instance of `ReentrantLock` with non-fair ordering policy—imposing no ordering for lock access. A non-fair lock may be acquired by a thread at the expense of other waiting reader and writer threads.
>
> Click here to view code image
>
> `ReentrantReadWriteLock(boolean fair)`
>
> Creates an instance of `ReentrantLock` with the given fairness policy. The value `true` indicates to use a fair ordering policy.

The basic idea behind a read-write lock is that a thread can use its read and write locks to perform thread-safe read and write operations on shared data, respectively. A thread is bound by the following lock acquisition rules for the read lock and the write lock of a read-write lock:

- *The read lock*: Only if no thread holds *the write lock* can a thread acquire the read lock. This means that the read lock can be shared by several reader threads— that is, a group of reader threads can access the shared data concurrently, as long as the write lock is not held by any thread.
- *The write lock*: Only if no thread holds *the write lock or the read lock* can a thread acquire the write lock. The write lock is thus mutually exclusive for writer threads—only one writer thread at a time can access the shared data.

Since the inner classes `ReadLock` and `WriteLock` implement the `Lock` interface, the lock acquisition policies of the `Lock` interface discussed earlier (p. 1462) are applicable to the read lock and the write lock, bearing in mind the specialized lock acquisition rules mentioned above for the read-write lock:

- *Unconditional locking* using the blocking `lock()` method
- *Polled locking* using the non-blocking `tryLock()` method
- *Timed locking* using the `tryLock(timeout, timeunit)` method
- *Interruptible locking* using the `lockInterruptibly()` method

As with any implementation of the `Lock` interface, the read lock and the write lock of a read-write lock must be released when done:

- *Released* using the `unlock()` method

Example 23.13 *Reentrant Read-Write Lock Counter*

Click here to view code image

```
package safe;
import java.util.OptionalInt;
import java.util.concurrent.locks.*;

public class ReentrantRWLockCounter implements ICounter {

  private ReadWriteLock rwl = new ReentrantReadWriteLock();           // (1)
  private Lock readLock = rwl.readLock();                             // (2)
  private Lock writeLock = rwl.writeLock();                           // (3)
  private int counter = 0;

  @Override
  public int getValue() {                                             // (4)
    readLock.lock();
    try {
//       System.out.println(Thread.currentThread().getName() + ": " + counter);
      return counter;
    } finally { readLock.unlock(); }
  }

  @Override
  public void increment() {                                           // (5)
    writeLock.lock();
    try {
      counter++;
    } finally { writeLock.unlock(); }
  }

  public int incrementAndGet() {                                      // (6)
    writeLock.lock();                 // Acquire write lock.
    try {
      return ++counter;                                               // (7)
//    ++counter;                      // Increment.                   // (8)
//    return getValue();              // Get the new value.           // (9)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public int getAndIncrement() {                                      // (10)
    writeLock.lock();                 // Acquire write lock.
    try {
      return counter++;               // Get and increment.           // (11)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public boolean incrIfPossible() {                                   // (12)
    if (writeLock.tryLock()) {          // Attempts to acquire the write lock.
      try {                             // Write lock acquired.
        counter++;
        return true;
      } finally { writeLock.unlock(); } // Write lock released.
    } else {                            // Write lock not acquired.
      return false;
    }
  }

  public OptionalInt getIfPossible() {                               // (13)
    if (readLock.tryLock()) {           // Attempts to acquire the read lock.
      try { return OptionalInt.of(counter); }
      finally { readLock.unlock(); }    // Read lock released.
    } else {                            // Read lock not acquired.
      return OptionalInt.empty();
    }
  }
}
```

Output from the program in Example 23.7, p. 1451:

Click here to view code image

```
Reentrant Read-Write Lock Counter: 10000
```

In Example 23.13, the class `ReentrantRWLockCounter` implements a thread-safe counter using a `ReentrantReadWriteLock`. It is a reworking of the counter in Example 23.12 that uses a `ReentrantLock`. In Example 23.13, the write lock allows mutually exclusive access to write operations and the read lock is shared concurrently by read operations. The numbered comments below correspond to the numbered code lines in Example 23.13.

(1)–(3) A non-fair `ReentrantReadWriteLock` is created, and its read and write locks obtained.

(4) The read lock acquired by a thread in the `getValue()` method ensures that the counter can be read safely, as no other thread can hold the write lock to modify the counter. This approach allows concurrent read operations and prevents concurrent write operations on the mutable counter, making it thread-safe. The read lock is guaranteed to be released in the `finally` block of the `try-finally` statement.

(5) The `increment()` method uses the write lock to provide exclusive access to increment the counter.

(6) The `incrementAndGet()` method shows that it is safe to increment *and* read the counter when the thread holds the write lock, as no other thread can access the counter. The `return` statement at (7) is equivalent to the statements at (8) and (9). The call to the `getValue()` method at (9) acquires the read lock to read the counter value, while the current write thread is still holding the write lock. This illustrates

the reentrant nature of the read-write lock, which allows a thread holding the write lock to acquire the read lock.

(10) The `getAndIncrement()` method shows that it is safe to read *and* increment the counter when the thread holds the write lock, as no other thread can access the counter.

(12) The `incrIfPossible()` method illustrates polling for the write lock. It uses the non-blocking `tryLock()` method in an attempt to acquire the write lock. If successful, it increments the counter, releases the write lock, and returns `true`. Otherwise, it returns `false`.

(13) The `getIfPossible()` method illustrates polling for the read lock. It uses the non-blocking `tryLock()` method in an attempt to acquire the read lock. If successful, it releases the read lock, and returns an `OptionalInt` with the value of the counter. Otherwise, it returns an empty `OptionalInt`, since the read lock could not be acquired.

Note that the methods in Example 23.13 implement *atomic actions*. When testing the `ReentrantRWLockCounter` class with Example 23.7, p. 1451, the output shows that the counter was incremented correctly by the threads.

The `ReentrantReadWriteLock` class also provides miscellaneous methods that can be used to query a read-write lock:

> ```
> boolean isFair()
> ```
>
> Determines whether this lock has its fairness policy set to `true`.
>
> ```
> int getReadHoldCount()
> int getWriteHoldCount()
> ```
>
> Queries the number of reentrant read holds and reentrant write holds on this lock by the current thread, respectively.
>
> Click here to view code image
>
> ```
> boolean isWriteLockedByCurrentThread()
> ```
>
> If the write lock is held by the current thread, returns `true`; otherwise, it returns `false`.

The fairness policy comes into play when the currently held lock is released and there are threads waiting for a lock. For a read-write lock, there may be both writer threads and reader threads waiting to acquire a lock. The following aspects of a read-write lock's fairness policy should be noted:

- In order to assign the read lock, the fairness policy basically chooses between *one longest-waiting writer thread* among any waiting writer threads to assign the write lock and *the longest-waiting group of reader threads* among any waiting reader threads. Whichever of these has waited the longest is chosen.
- A thread can only acquire a fair read lock if both the write lock is free and there are no waiting writer threads. The thread will only acquire the read lock after the oldest currently waiting writer thread has acquired and released the write lock.
- A thread can only acquire a fair write lock if both the read lock and write lock are free; otherwise, it will block.

Analogous to the `tryLock()` methods of the `ReentrantLock` class, the `tryLock()` methods of the inner classes `ReadLock` and `WriteLock` do not honor the fairness policy, acquiring the lock immediately if available, regardless of any waiting threads.

Analogous to the reentrant nature of a `ReentrantLock`, the read-write lock allows both reader and writer threads to reacquire its read lock or write lock. Note that a thread that holds the write lock can acquire the read lock, but not vice versa. A reader thread that tries to acquire the write lock will block and never succeed.

Although a `ReentrantReadWriteLock` allows a greater level of concurrency to access shared data than a `ReentrantLock`, many factors can influence the performance of a read-write lock. Profiling is recommended to gauge the impact of such factors as its premises of more frequent reads than writes, duration of read and write operations, and number of cores available to leverage parallel execution. However, other thread-safe solutions provided by the Concurrency API for accessing shared data should also be considered.

23.5 Special-Purpose Synchronizers

The `java.util.concurent` package provides classes that implement useful special-purpose synchronization idioms that impose specific behavior on how threads can collectively make progress through some algorithm. We consider two such classes in this section:

`CyclicBarrier` and `CountDownLatch`.

Cyclic Barrier

The `java.util.concurrent.CyclicBarrier` class implements a *barrier* upon which a set of threads have to synchronize before they are all allowed to proceed. The threads may need to wait for one another until all threads reach the barrier. When this happens the barrier will *trip*, allowing them to continue. The barrier is *cyclic*, as the same barrier can be reused after the waiting threads have been released.

The following constructors of the `CyclicBarrier` class can be used to create a barrier.

> Click here to view code image
>
> ```
> CyclicBarrier(int parties)
> CyclicBarrier(int parties, Runnable barrierAction)
> ```
>
> Both methods create a barrier that will trip when the specified number of `parties` (i.e., threads) are waiting for it—that is, when these invoke the `await()` method on it.
> The barrier created by the second method will execute the given *barrier action* when the barrier is tripped. This action is performed by the *last* thread arriving at the barrier.

The declaration at (1) below creates a barrier that will trip when three threads synchronize on it. The declaration at (2) creates a barrier that will trip when five threads are waiting for it. In addition, a *barrier action* is defined that will be performed when the barrier is tripped. Typically, a barrier action can be used for consolidating any results computed by the threads thus far.

Click here to view code image

```
CyclicBarrier barrier1 = new CyclicBarrier(3);                    // (1)
CyclicBarrier barrier2 = new CyclicBarrier(5, () ->
        System.out.println("All results accumulated."));          // (2)
```

A cyclic barrier is *shared* by the threads wishing to synchronize on it. When a thread wants to wait at the barrier, it calls the `await()` method on the barrier. This defines a *barrier point*. The call blocks at the barrier point until the number of threads required to trip the barrier is reached—that is, the remaining threads also invoke the `await()` method. Otherwise, the barrier will not trip as long as the number of waiting threads is less than the number required by the barrier.

Click here to view code image

```
// Thread t1 waits on barrier1 at this barrier point.
barrier1.await();           // Can continue when the barrier is tripped.
...
// Thread t2 waits on barrier1 at this barrier point.
barrier1.await();           // Can continue when the barrier is tripped.
...
// Thread t3 waits on barrier1 at this barrier point.
barrier1.await();           // Can continue when the barrier is tripped.
...
```

The barrier `barrier1` is tripped when any three threads invoke the `await()` method on this barrier—for example, threads `t1`, `t2`, and `t3`, as shown above. When tripped, the waiting threads can proceed, and the cyclic barrier resets itself and can be tripped again.

The `CyclicBarrier` class also defines methods to query and reset a barrier.

> Click here to view code image
>
> ```
> int await() throws InterruptedException, BrokenBarrierException
> int await(long timeout, TimeUnit unit) throws InterruptedException,
>                                               BrokenBarrierException,
>                                               TimeoutException
> ```
>
> Both methods wait until all parties have invoked the `await()` method on this barrier, or until the specified waiting time elapses, as in the second method.
>
> These methods return the *arrival index* of the current thread, index *N*`-1` for the first thread to arrive and `0` for the last thread to arrive, where *N* is the number of parties required to trip this barrier.
>
> An `InterruptedException` is thrown if the current thread was interrupted while waiting.
>
> A `BrokenBarrierException` is thrown if another thread was interrupted or timed out while the current thread was waiting, or the barrier was reset, or the barrier was broken when the `await()` method was called, or any barrier action failed due to an exception.
>
> A `TimeoutException` is thrown if any specified timeout elapses. The barrier is then broken.
>
> ```
> int getParties()
> ```
>
> Returns the number of parties required to trip this barrier.
>
> ```
> boolean isBroken()
> ```

> Queries whether this barrier is in a *broken state*. A barrier can break due to an interruption, a timeout, the last reset, or a failed barrier action due to an exception. It returns `true` if any of the parties broke the barrier; otherwise, it returns `false`.
>
> `void reset()`
>
> Resets the barrier to its initial state—that is, the wait count of the barrier is set to 0. Any parties currently waiting at this barrier will return with a `Broken-BarrierException`.

Example 23.14 illustrates using a single cyclic barrier that is shared by three threads. The barrier is declared at (2), requiring three parties and having the barrier action at (1) being performed when the barrier is tripped. The task declared at (3) calls the `await()` method on the barrier when it wants to synchronize on the barrier. Note the checked exceptions that can be thrown by the `await()` method. The threads are created and started at (4).

The output from Example 23.14 shows that each thread waited at the barrier point and announced it was released after the barrier tripped. The barrier action is performed by thread `T2` that was the last one to arrive at the barrier point.

Example 23.14 *Cyclic Barrier*

Click here to view code image

```
package synchronizers;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
  public static final int PARTIES = 3;

  public static void main(String args[]) {
    Runnable barrierAction =  () ->                                       //
          System.out.println("Barrier action by "
                               + Thread.currentThread().getName()
                               + ": All tasks are released.");

    CyclicBarrier barrier = new CyclicBarrier(PARTIES, barrierAction);    //

    Runnable task = () -> {                                               //
      String threadName = Thread.currentThread().getName();
      try {
        System.out.println(threadName + " is now waiting");
        barrier.await();                                       // Barrier poi
        System.out.println(threadName + " is now released");
      } catch (BrokenBarrierException | InterruptedException e) {
        e.printStackTrace();
      }
    };
```

```
      for (int i = 0; i < PARTIES; i++) {                                    //
        new Thread(task, "T" + (i+1)).start();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
T1 is now waiting
T3 is now waiting
T2 is now waiting
Barrier action by T2: All tasks are released.
T2 is now released
T1 is now released
T3 is now released
```

Threads can of course share more than one barrier, having a different number of parties and different barrier actions.

Once a barrier is tripped, its thread wait count is reset to zero, and the barrier can be used again to implement a barrier point by calling the `await()` method.

If more threads than the required number invoke the `await()` method, then the required number of threads will cause the barrier to trip. This will allow the required number to continue and will reset the wait count, and the extra threads will have to wait for the barrier to trip again.

The cyclic barrier implements the *all-or-none model*. Either all waiting threads successfully continue execution past the `await()` method, or none do. If one thread waiting at a barrier point leaves prematurely—which could be due to an interruption, timeout, or failure—then all other waiting threads also leave abnormally via the checked `java.util.concurrent.BrokenBarrierException`. In this case, the barrier is said to be *broken*. The method `isBroken()` can be used to determine whether this is the case. A broken barrier can be reset and reused.

Count-Down Latch

The `java.util.concurrent.CountDownLatch` class implements a *count-down latch* that allows one or more threads to wait until the latch goes up, and then the waiting threads are allowed to proceed. The latch goes up depending on one or more other threads performing a specific number of countdown operations.

The `CountDownLatch` class provides the following constructor to create a count-down latch.

> ```
> CountDownLatch(int count)
> ```
>
> Creates a `CountDownLatch` initialized with the given `count`. The `count` specifies the number of times the `countDown()` method must be invoked to raise the latch to allow threads waiting on the `await()` method to pass through.

A count-down latch is created with an initial count. One or more threads can call the `await()` method of the latch to wait until the count reaches zero. The count is decremented by calls to the `countDown()` method of the latch by one or more other threads. Note that it is the number of calls to the `countDown()` method that raises the latch, independent of the number of threads involved. The `countDown()` method is used in tandem with the `await()` method to operate the latch. The latch can only go up once, and therefore cannot be reused.

> Click here to view code image
>
> ```
> void await() throws InterruptedException
> boolean await(long timeout, TimeUnit unit) throws InterruptedException
> ```
>
> Both methods cause the current thread to wait until one of the following events occurs: The latch count reaches 0, the thread is interrupted, or if any specified waiting time has elapsed.
> In the first method, it is possible that the current thread may wait forever, if the count never reaches 0.
> The second method returns `true` if the count reached 0 and `false` if the waiting time elapsed before the count reached 0.
>
> ```
> void countDown()
> ```
>
> Decrements the current count of the latch if it is greater than 0, releasing all waiting threads if the new count reaches 0.
> Nothing happens if the current count is already 0.

Given that the latch below is shared by threads `t1`, `t2`, and `t3`:

Click here to view code image

```
CountDownLatch latch = new CountDownLatch(3);    // (1) Latch count is 3.
```

Threads `t1` and `t2` call the `await()` method and wait at (2) and (3), respectively, for the latch to be raised:

Click here to view code image

```
// Thread t1:
latch.await();              // (2) Waiting for latch to be raised.
...
// Thread t2:
latch.await();              // (3) Waiting for latch to be raised.
...
```

Thread t3 calls the `countDown()` method on the latch, decrementing the count each time. Note that the thread calling the `countDown()` method does *not* wait for the count to reach 0. The count reaches 0 when the call at (4) is executed, upon which the latch is raised, and waiting threads `t1` and `t2` can continue execution past the `await()` method call.

Click here to view code image

```
// Thread t3:
latch.countDown();
...
latch.countDown(),
...
latch.countDown();         // (4) Count reaches 0. Latch released at (2) and (
...
```

Example 23.15 illustrates a typical scenario where an administrator thread (in this case, the main thread) submits tasks (defined by the `Task` class at (7)) to individual threads and uses two count-down latches to start and wait for the tasks to finish.

The two latches, `startLine` and `finishLine`, are declared at (1) and (2), respectively. The `startLine` latch is an *on/off latch* with a count of 1. All submitted tasks invoke the `await()` method at (9) to wait until the `startLine` latch is raised by the main thread by invoking the `countDown()` method at (5) to decrement the count to 0.

The `finishLine` latch is a latch with a count of *N*. It is the converse of the `startLine` latch. All submitted tasks count down the `finishLine` latch by invoking the `count-Down()` method at (10) and continue execution. The main method invokes the `await()` on the `finishLine` latch and waits until the latch is raised by all submitted tasks, decrementing the count to 0.

Example 23.15 *Count-Down Latch*

Click here to view code image

```
package synchronizers;
import static java.lang.System.out;
import java.util.concurrent.*;
```

```
public class CountDownLatchTest {
  public static final int N = 3;

  public static void main(String[] args) throws InterruptedException {
    CountDownLatch startLine = new CountDownLatch(1);              // (1)

    CountDownLatch finishLine = new CountDownLatch(N);             // (2)

    ExecutorService es = Executors.newFixedThreadPool(N);          // (3)
    String threadName = Thread.currentThread().getName();
    try {
      for (int i = 0; i < N; ++i) {    // (4) Submit tasks.
        es.submit(new Task(startLine, finishLine));
      }
      out.println(threadName + ": Let all tasks proceed.");
      startLine.countDown();           // (5) Count down to let all tasks proc
      finishLine.await();              // (6) Wait for all tasks to finish.
      out.println(threadName + ": All tasks done.");
    } finally {
      es.shutdown();
    }
  }
}

class Task implements Runnable {                                   // (7)
  private final CountDownLatch startLine;
  private final CountDownLatch finishLine;

  public Task(CountDownLatch start, CountDownLatch finish) {
    this.startLine = start;
    this.finishLine = finish;
  }

  @Override
  public void run() {                                               // (8)
    String threadName = Thread.currentThread().getName();
    try {
      out.println(threadName + ": Waiting to proceed.");
      startLine.await();               // (9) Wait to proceed.
      out.println(threadName + ": Running ... ");
      finishLine.countDown();          // (10) Count down the latch & continue
      out.println(threadName + ": Latch count decremented.");
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
main: Let all tasks proceed.
pool-1-thread-3: Waiting to proceed.
pool-1-thread-2: Waiting to proceed.
pool-1-thread-1: Waiting to proceed.
pool-1-thread-1: Running ...
pool-1-thread-2: Running ...
pool-1-thread-2: Latch count decremented.
pool-1-thread-3: Running ...
pool-1-thread-1: Latch count decremented.
pool-1-thread-3: Latch count decremented.
main: All tasks done.
```

23.6 Synchronized Collections and Maps

The primary goal of the Collections Framework is to provide support for efficient computation on collections that can be used in single-threaded applications. In the rest of this chapter we explore the support provided by the Collections Framework and the Concurrency Framework to create thread-safe collections that can be used in multithreaded applications. We start with *synchronized collections* provided by the `java.util` package, and explore *concurrent collections* found in the `java.util.concurrent` package in the next section (p. 1482).

By a *thread-safe collection* we mean a collection whose elements can be processed by concurrent threads without ever being in an inconsistent state.

Unmodifiable collections cannot be changed structurally (add and remove operations are prohibited) and their elements cannot be replaced (throw an `Unsupported-OperationException`), making the unmodifiable collection immutable (§12.2, p. 649). This guarantees that an unmodifiable collection is thread-safe, and multiple reader threads can safely access the elements concurrently as long as it is done via the unmodifiable collection.

Click here to view code image

```
// Thread-safe unmodifiable list:
List<String> list = List.of("Tom", "Dick", "Harriet");
list.add("Harry");                          // UnsupportedOperationException
list.set(0, "Tommy");                       // UnsupportedOperationException
list.remove("Dick");                        // UnsupportedOperationException
System.out.println(list.get(0));            // "Tom"
```

Unmodifiable views of collections are backed by an underlying collection, where changes in the underlying collection are reflected in the view (§15.11, p. 856). Such a view also cannot be modified, and only query methods are passed to the underlying collection. The view can be considered effectively immutable, and thus thread-safe, if the backing collection is effectively immutable, or if the only reference to the backing collection is through the unmodifiable view. An example of such an unmodifiable view is given below:

Click here to view code image

```
// Effectively immutable list view:
List<Integer> list = new ArrayList<>();
list.add(2021); list.add(2022);
List<Integer> immutablelist = Collections.unmodifiableList(list);
immutablelist.add(2023);                       // UnsupportedOperationExceptio
immutablelist.set(0, 2023);                    // UnsupportedOperationExceptio
immutablelist.remove(2021);                    // UnsupportedOperationExceptio
System.out.println(immutablelist.get(0));      // 2021
```

It is worth keeping in mind that a thread-safe collection does not imply that its elements are thread-safe. It might be necessary to employ an appropriate synchronization mechanism on the elements in order to modify them in a thread-safe way.

The `java.util.Collections` class provides methods to create *synchronized collections* that are thread-safe. Actually, these collections are *synchronized views of collections* as they are backed by an *underlying collection*. The methods provided accept a collection and return a view of this collection in which the getter and setter methods use synchronized blocks to delegate operations to the underlying collection—that is, the methods provide a synchronized wrapper around the underlying collection. A single intrinsic lock on the synchronized collection implements mutual exclusion of operations, but this can potentially become a point of contention among threads. Synchronized collections also incur the performance penalty associated with intrinsic locking. Other thread-safe solutions explored in this chapter should be considered before settling on using a synchronized view to make a collection thread-safe.

Click here to view code image

```
static <E> Collection<E>    synchronizedCollection(Collection<E> c)
static <E> List<E>          synchronizedList(List<E> list)
static <E> Set<E>           synchronizedSet(Set<E> set)
static <E> SortedSet<E>     synchronizedSortedSet(SortedSet<E> set)
static <E> NavigableSet<E> synchronizedNavigableSet(NavigableSet<E> set)
```

Return a synchronized (thread-safe) *view* collection that has the same type as the specified backing collection. It is imperative that any access to the backing collection is accomplished through the returned collection (or its views) to guarantee serial access.

Analogous to an unmodifiable view collection (§15.11, p. 856), the synchronized collection returned by the `synchronizedCollection()` method does not delegate the `equals()` and the `hashCode()` methods to the backing collection. Instead, the returned collection uses the corresponding methods inherited from the `Object` class. This is to safeguard the contract of these methods when the backing collection is a set or a list. However, the synchronized collections returned by the other methods above do not

> exhibit this behavior.
>
> Click here to view code image
>
> ```
> static <K,V> Map<K,V>       synchronizedMap(Map<K,V> map)
> static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> map)
> static <K,V> NavigableMap<K,V>
>                             synchronizedNavigableMap(NavigableMap<K,V> map)
> ```
>
> Return a synchronized (thread-safe) *view* map that has the same type as the backing map that is specified. It is imperative that any access to the backing map is accomplished through the synchronized map (e.g., its key, entry, or values views) to guarantee thread-safe serial access.

We use the synchronized list as an example in this section. For using the other synchronized collections and maps, familiarity with their unsafe counterparts will go a long way. For the most part, the synchronized collections provide mutually exclusive operations corresponding to the operations on their unsafe counterparts.

Serial Access

In order to guarantee thread-safety, *all* access to elements of the underlying collection should be through the synchronized collection. Although getter and setter methods are synchronized in a synchronized collection, this is *not* the case for methods that implement *serial access* (e.g., iteration using an iterator or a stream).

As multiple methods can be called for an operation on a synchronized collection when using the iterator, we need to ensure that these are executed as a single mutually exclusive operation. Iteration thus requires a coarse-grained manual synchronization on the synchronized collection for deterministic results. A more general case of coarse-grained synchronization is discussed in the next subsection (p. 1480).

Example 23.16 illustrates the idiom used for serial access over a synchronized collection. A synchronized list is created at (1) and populated. A `Runnable` is implemented at (2) to remove a certain year from the synchronized list.

Serial access over the synchronized list is attempted using an explicit iterator. However, serial access operations are all done in a synchronized block requiring the intrinsic lock on the synchronized list, as at (3). Obtaining the iterator is also done in the synchronized block. All modifications must be made through the iterator. Iterator methods are used to process the synchronized list, including removing elements. Keep in mind that `next()` and `remove()` methods of the iterator work in lockstep. The current element from each iteration can be processed safely. This idiom also applies when using a `for(:)` loop for serial access, as this loop internally is translated to an iterator.

The program in Example 23.16 is run with three threads that execute the `Runnable eliminator`. With manual synchronization on the synchronized list at (3), the threads execute as expected. One of the threads removed the year 2021 from the list and thus modified the list. As there was only one occurrence of the year 2021, the other two threads did not modify the list. With no manual synchronization on the synchronized list, the results are unpredictable. Most likely one or more exceptions will be thrown.

Regardless of manually synchronizing on the synchronized collection, as in Example 23.16, any modification made directly on the *underlying* collection during serial access will result in the runtime `java.util.ConcurrentModificationException`.

Example 23.16 *Serial Access in Synchronized Views of Collections*

Click here to view code image

```
package synced;
import java.util.*;
import java.util.stream.IntStream;

public class SerialAccessThreads  {

  public static void main(String[] args) throws InterruptedException {
    List<Integer> years = Collections.synchronizedList(new ArrayList<>()); //
    years.add(2024); years.add(2023); years.add(2021); years.add(2022);

    Runnable eliminator = () -> {                                          //
      boolean found = false;
      synchronized(years) {                                                //
        Iterator<Integer> iteratorA = years.iterator();                    //
        while (iteratorA.hasNext()) {
          if (iteratorA.next().equals(2021)) {                             //

            iteratorA.remove();                                            //
            found = true;
          }
        }
      }                                                                    //
      System.out.println("List modified: " + found);
    };
    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(eliminator).start());
  }
}
```

Probable output from the program with manual synchronization for serial access:

```
List modified: false
List modified: true
List modified: false
```

Probable output from the program without manual synchronization for serial access (comment out lines at (2) and (7)) (*output edited to fit on the page*):

Click here to view code image

```
Exception in thread "Thread-1"
Exception in thread "Thread-2"
Exception in thread "Thread-0" java.lang.NullPointerException:
    Cannot invoke "java.lang.Integer.equals(Object)" because the return value
    "java.util.Iterator.next()" is null
    ...
```

Compound Mutually Exclusive Operations

A compound operation that requires multiple operations on a synchronized collection is not guaranteed to be thread-safe just because the individual operations are mutually exclusive. Concurrent threads executing such a compound operation can interleave the individual operations, as in the case of a compound arithmetic operator. Example 23.17 illustrates implementing a compound mutually exclusive operation on a synchronized list using coarse-grained synchronization.

The class `DoubleAct` has a synchronized list (`names`) declared at (1), and provides two methods `add()` and `removeFirst()` at (2) and (3) to maintain this list.

The `Client` class uses the `DoubleAct` class. An instance of the `DoubleAct` class is populated at (10) in the `main()` method. A `Runnable` (`remover`) is implemented by the lambda expression at (11). It calls the `removeFirst()` method at (12) and prints the retrieved name. Three threads are created at (13) that execute the `Runnable remover`, calling the `removeFirst()` method on the shared `DoubleAct` instance. Since there are only two elements in the list, two of the threads should print one name each, and one of them should return `null`.

The method `removeFirst()` at (3) uses two mutually exclusive methods of the synchronized list. The intent is to return the element at index 0 if the list is not empty; otherwise, the value `null`. The method `size()` returns the current size of the synchronized list, and the method `remove()` deletes the element at index 0. The calls to these methods can be interleaved by concurrent threads executing the `removeFirst()` method between the time the list size is determined and the first element is removed, unless appropriate steps are taken.

Click here to view code image

```
// Thread t1                                  // Thread t2
...                                           ...
if (name.size() > 0) // true                  ...

...                                           if (name.size() > 0) // true

return remove(0);    // Size 0                ...

...                                           return remove(0);    // Exception
```

The recommended solution is to use a synchronized block on the synchronized list, as shown at (4), to guarantee that the method `removeFirst()` has exclusive access to the synchronized list, until the individual exclusive operations have completed. Example 23.17 shows output for both scenarios that we have sketched above. Note that the reentrant nature of the intrinsic lock on the synchronized list allows nested locking on the list for the individual exclusive operations.

Example 23.17 *Compound Mutually Exclusive Operation*

Click here to view code image

```
package synced;
import java.util.*;

public class DoubleAct {
  // Synchronized list:                                                        (
  private List<String> names = Collections.synchronizedList(new ArrayList<>())

  public void add(String name) { names.add(name); }                        //  (

  public String removeFirst() {                                            //  (
    synchronized(names) {                                                  //  (
      if (names.size() > 0) {                                              //  (
        try { Thread.sleep(1); }                                           //  (
        catch(InterruptedException e) { e.printStackTrace(); }
        return names.remove(0);                                            //  (
      } else { return null; }                                              //  (
    }                                                                      //  (
  }
}
```

Click here to view code image

```
package synced;
import java.util.stream.IntStream;

public class Client {                                                      //  (
  public static void main(String[] args) {
    DoubleAct da = new DoubleAct();
    da.add("Laurel"); da.add("Hardy");                                    // (
    Runnable remover = () -> {                                            // (
      String name = da.removeFirst();                                     // (
      System.out.println(name);
    };

    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(remover).start());// (
  }
}
```

Probable output from the program:

```
Laurel
Hardy
null
```

Probable output from the program when (4) and (9) are commented out (*output edited to fit on the page*):

Click here to view code image

```
Laurel
Hardy
Exception in thread "Thread-1" java.lang.IndexOutOfBoundsException:
      Index 0 out of bounds for length 0
      at ...
      at synced.DoubleAction.removeFirst(DoubleAction.java:15)
      at ...
```

23.7 Concurrent Collections and Maps

Most of the collections in the `java.util` package are not thread-safe—executing concurrent operations on them is courting disaster. Exceptions to this are the legacy `Vector` and `Hashtable` classes, but these use a single lock, allowing only one thread at a time to execute an operation on the collection while other threads are blocked. The *synchronized collections* created by methods in the `java.util.Collections` class are not much better, as they are just thread-safe wrappers around a backing collection, again providing only mutually exclusive operations on the backing collection. However, the *concurrent collections* provided by the `java.util.concurrent` package use special-purpose locking mechanisms to enable multiple threads to operate concurrently, without explicit locking on the collection and with minimal contention. The concurrent collections offer greater flexibility and higher scalability compared to the collections in the `java.util` package, when concurrent access is crucial for multiple threads sharing objects stored in a collection.

We assume familiarity with the Collections Framework, especially the core interfaces in the `java.util` package (Chapter 15, p. 781), as these are implemented by the concurrent collections to provide thread-safety and atomicity of operations defined by these interfaces:

`Q(§e1l5e.6t,ipn 873)`
`D(§e1q5u.e3, p. 864)`
`M(§a1p5.4, p. 897)`
`N(§a1v5i.g1a0b,lpeMpa.p9 909))`	

It is also important to note that concurrent collections avoid memory consistency errors by defining a happens-before relationship that essentially states that an action by a thread to place an object into a concurrent collection happens-before a subsequent action to access or remove that object from the collection by another thread.

The concurrent collections and maps in the `java.util.concurrent` package can be categorized as follows:

- *Concurrent collections (p. 1485)*
- *Concurrent maps (p. 1490)*
- *Blocking queues (p. 1495)*
- *Copy-on-write collections (p. 1501)*

The following aspects about collections and maps should be noted, as they can be crucial in selecting an appropriate collection for maintaining the elements:

- *Non-blocking or blocking*

  Operations on a *non-blocking collection* do not use any locking mechanism (are *lock-free*) to access elements of the collection, allowing multiple threads to execute concurrently, in contrast to a *blocking collection* in which operations can block to acquire a lock before they can proceed, because only one thread at a time is allowed.

  Collections with `Concurrent` as a prefix to their name are all non-blocking—for example, a `ConcurrentHashMap`. All queues implementing the `BlockingQueue<E>` interface, not surprisingly, are blocking collections. The names of many queues reveal their blocking nature—for example, a `LinkedBlockingQueue`.

- *Unbounded and bounded*

A *bounded collection* has a *fixed capacity* that defines the maximum number of elements allowed in the collection and is usually specified when the collection is created. When the collection is full, some operations, such as adding an element, will block until there is space in the collection. *Unbounded collections* are not bounded by any capacity restriction. *Optionally bounded collections* can be instantiated to behave either as bounded or as unbounded.

An `ArrayBlockingQueue` is a bounded queue, whereas a `ConcurrentLinkedQueue` is unbounded, but a `LinkedBlockingDeque` can be optionally bounded.

- *The* `null` *value*

  Some collections allow the `null` value, and others do not. All concurrent collections, blocking queues, and concurrent maps do *not* allow the `null` value as elements, whereas the copy-on-write collections do.

- *Duplicate elements*

  All collections that embody the concept of a set do not allow duplicates—for example, a `CopyOnWriteArraySet`. All maps (e.g., a `ConcurrentHashMap`) do not allow duplicate keys which would violate the concept of hashing values. However, all queues allow duplicates—for example, a `LinkedBlockingDeque`.

- *Ordering*

  Elements in some concurrent sets and maps do not have any ordering—for example, a `CopyOnWriteArraySet` or a `ConcurrentHashMap`.

  Some sets and maps keep their elements and entries in *sort order*, according to either their natural ordering or a total ordering defined by a comparator on the elements or the keys, respectively. Examples of concurrent sorted collections are a `ConcurrentSkipListSet` and a `ConcurrentSkipListMap`.

  Insertion ordering is the order in which the elements are inserted into the collection—as used by a `CopyOnWriteArrayList` for its elements.

  FIFO (*First-In-First-Out*) order is maintained by some thread-safe queues, such as a `LinkedBlockingQueue` and a `ConcurrentLinkedQueue`. Other queues have a more specialized ordering, like a `DelayQueue`.

Not surprisingly, some thread-safe deques exhibit both FIFO and LIFO (*Last-In-First-Out*) order for their elements depending on how the deque is used—for example, a `LinkedBlockingDeque`.

- *Iterator behavior during serial access*

  Serial access is implemented differently for different categories of concurrent collections. It is important to understand the behavior of the iterator provided by these collections. An iterator has to address situations where modifications are made to the collection, outside the control of the iterator, that might render the state of the collection inconsistent—for example, removing or inserting elements during iteration. Iterators can be classified according to the response they provide in this situation, which is referred to as *concurrent modification*:

  - *Weakly consistent iterator*

    Most of the concurrent collections, and especially queues, provide an iterator that is *weakly consistent* during serial access. Such an iterator is created on a clone of the collection, and will always traverse elements that existed at the time the iterator was constructed only once. It *may not* reflect all subsequent modifications made to the collection after the iterator is constructed. It will also not throw a `ConcurrentModificationException`, and can execute concurrently with other operations on the collection. Weakly consistent iterators also provide guarantees against repeated elements occurring, and guard against a variety of errors and from infinite loops occurring during traversal.

    Weakly consistent iterators are also informally referred to as *fail-safe* iterators.

  - *Snapshot-style iterator*

    A `CopyOnWriteArrayList`, which is based on a thread-safe version of an `ArrayList`, uses *snapshot-style iterators* for serial access. The name of the iterator reflects that it iterates over a snapshot of the underlying array taken at the time the iterator is created. Concurrent multiple *read* operations on this array snapshot are thread-safe as this snapshot of the array cannot be changed, but any *write* operation is done on a *new copy* of the array. An iterator is always created on the most current modified array. A snapshot iterator does *not* reflect any changes made in the copy of the array—in contrast to a weakly consistent iterator that may reflect such changes.

- *Fail-fast iterator*

  At the start of each iteration, a fail-fast iterator detects whether the collection has been modified. If that is the case, it throws a `ConcurrentModification-Exception`. In other words, it fails as soon as possible at the next iteration. It is perfectly safe during iteration to use the `Iterator.remove()` method to delete the current element, but not with the `Collection.remove()` method.

  Thread-unsafe collections from the `java.util` package exhibit this behavior, as do the `PriorityBlockingQueue` and `DelayDeque` classes in the `java.util.concurrent` package.

Concurrent Collections

Concurrent sets, queues, and deques are implemented by the classes `Concurrent-SkipListSet`, `ConcurrentLinkedQueue`, and `ConcurrentLinkedDeque`, respectively. These thread-safe classes implement the corresponding `NavigableSet`, `Queue`, and `Deque` interfaces in the `java.util` package, as shown in Figure 23.4 and summarized in Table 23.5. Their characteristics are summarized in Table 23.6. These concurrent collections are *unbounded* and *non-blocking*. They do not allow `null` values and have a weakly consistent iterator. As they are unbounded, their insert operation will always succeed. Also worth noting is that their *bulk operations* (`addAll()`, `removeIf()`, `forEach()`) are not guaranteed to perform atomically.

Figure 23.4 *Concurrent Collections in the* `java.util.concurrent` *Package*

Table 23.5 *Concurrent Collections in the* `java.util.concurrent` *Package*

Concurrent collections	Description

`ConcurrentSkipListSet<E> implements NavigableSet<E>`	An unbounded, non-blocking, concurrent `NavigableSet` implementation that internally uses a `ConcurrentSkipListMap`. Elements are sorted according to the natural ordering or by a comparator that is provided in the constructor.
`ConcurrentLinkedQueue<E> implements Queue<E>`	An unbounded, non-blocking, concurrent queue based on linked nodes.
`ConcurrentLinkedDeque<E> implements Deque<E>`	An unbounded, non-blocking, concurrent deque based on linked nodes.

Table 23.6 *Characteristics of Concurrent Collections*

Concurrent collections	Duplicates	Ordering	Null values	Kind of iterator
`ConcurrentSkipListSet<E> implements NavigableSet<E>`	No	Sorted	No	Weakly consistent
`ConcurrentLinkedQueue<E> implements Queue<E>`	Yes	FIFO	No	Weakly consistent
`ConcurrentLinkedDeque<E> implements Deque<E>`	Yes	FIFO/LIFO	No	Weakly consistent

The `ConcurrentSkipListSet<E>` *Class*

The `ConcurrentSkipListSet` class implements the `NavigableSet` interface in the `java.util` package (§15.5, p. 811). The class implements a scalable, concurrent, sorted set. It provides efficient insertion, removal, and access operations that can be executed safely and concurrently by multiple threads. Its salient properties are summarized in Table 23.6. Selected methods provided by the `ConcurrentSkipListSet` class are listed below.

```
// First-last elements
E pollFirst()
E pollLast()
```

Remove the first and the last elements currently in this concurrent set, respectively.

Click here to view code image

```
// Range-view operations
NavigableSet<E> headSet(E toElement,   boolean inclusive)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

Return different views of the underlying concurrent set, depending on the bound elements.

```
// Closest-matches
E ceiling(E e)
E floor(E e)
E higher(E e)
E lower(E e)
```

Determine closest-match elements according to various criteria.

Click here to view code image

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

The first method returns a reverse-order iterator for this concurrent set. The second method returns a reverse-order view of the elements in the set.

The **`ConcurrentLinkedQueue<E>`** *Class*

The `ConcurrentLinkedQueue` class implements the `Queue` interface, as shown in Figure 23.6. No new methods are defined, and existing methods of the `Queue` interface (§15.6, p. 814) are implemented as shown in Table 23.7. Note that since it is an unbounded queue, the insert operations will always succeed.

The **`ConcurrentLinkedDeque<E>`** *Class*

The `ConcurrentLinkedDeque` class implements the `Deque` interface, as shown in Figure 23.6. No new methods are defined, and existing methods of the `Deque` interface (§15.7, p. 821) are implemented as shown in Table 23.8. Note that since it is an unbounded deque, the insert operations will always succeed. Methods inherited from the `Queue` interface are marked with an asterisk (`*`) in Table 23.8.

Table 23.7 *Selected Methods in the* `ConcurrentLinkedQueue` *Class*

Operation	**Throws exception**	**Returns special value**
Insert at the tail	`add(e)`, *will either throw* IllegalArgumentException	`offer(e)`, *will either return*
Remove from the head	`remove()`, *throws* `NoSuchElementException`	`poll()`, *returns* `null` *if empty*
Examine element at the head	`element()`, *throws* `NoSuchElementException`	`peek()`, *returns* `null` *if empty*

Table 23.8 *Selected Methods in the* `ConcurrentLinkedDeque` *Class*

Insert at the head	**Insert at the tail**	**Runtime behavior on failure**
`offerFirst(e)`	`offerLast(e)`, `offer(e)*`	*Never returns* `false`
`addFirst(e)`	`addLast(e)`, `add(e)*`	*Never throws* `IllegalStateException`
Remove from the head	**Remove from the tail**	**Runtime behavior on failure**
`pollFirst()`	`pollLast()`, `poll()*`	*Returns* `null`, *if empty*
`removeFirst()`	`removeLast()`, `remove()*`	*Throws* `NoSuchElementException`
Examine at the head	**Examine at the tail**	**Runtime behavior on failure**
`peekFirst()`	`peekLast()`, `peek()*`	*Returns* `null`, *if empty*
`getFirst()`	`getLast()`, `element()*`	*Throws* `NoSuchElementException`

Example 23.18 illustrates the iterator of a `ConcurrentSkipListSet` (declared at (1)) that is weakly consistent during traversal. The idea is to have two tasks operating on the concurrent sorted set: one repeatedly creating an iterator to traverse the elements of the collection and summing them (2), and another continuously removing elements from the set (5).

The utility class `ConcUtil` declares the auxiliary method `snooze()` that is used to put a thread to sleep. If interrupted, the exception is caught and the interrupt reinstated so that the caller of the `snooze()` method can take the appropriate action. This method is also used in other examples in this section.

The `Runnable sumValues` at (2) repeatedly creates an iterator at (3) to sum the values in the set at the time the iterator is created, and prints the result. It snoozes a little after each traversal of the set. The thread determines at (4) whether it has been interrupted. If interrupted, the infinite loop terminates, thereby terminating the thread.

The `Runnable removeValues` at (5) polls the set and prints the value, snoozing after each polling operation. It also detects whether it has been interrupted, terminating the infinite loop at (6) if that is the case.

The `main()` method instantiates and initializes a `ConcurrentSkipListSet` at (7) with random numbers between 0 and 1000. The two tasks are submitted to a service executor at (8). The main thread snoozes a little to allow the tasks to run. The call to the `shutdownNow()` method at (10) leads to any thread running to be interrupted, and the threads taking appropriate action to terminate their execution as described above.

The output from the program shows that only elements that are in the `Concur-rentSkipListSet` instance at the time the iterator is created contribute to the sum, and the sum drops correctly as elements are removed.

Example 23.18 *Concurrent Collections*

Click here to view code image

```
package concurrent;
import java.util.concurrent.TimeUnit;

public class ConcUtil {
  public static void snooze(int timeout, TimeUnit unit) {
    String threadName = Thread.currentThread().getName();
    try {
      unit.sleep(timeout);
    } catch (InterruptedException ex) {
      System.out.println(threadName + ": " + ex);
      Thread.currentThread().interrupt();          // Reinstate interrupt statu
    }
  }
}
```

Click here to view code image

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class ConcurrentSkipListSetDemo {

  private static ConcurrentSkipListSet<Integer> set;                       // (

  private static Runnable sumValues = () -> {                              // (
    String threadName = Thread.currentThread().getName();
    while (true) {
      int sum = 0;
      for (Integer v : set) {                                              // (
        sum += v;
      }
      System.out.printf(threadName + ": sum%9d%n", sum);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
      if (Thread.interrupted()) break;                                     // (
    }
  };

  private static Runnable removeValues = () -> {                           // (
    String threadName = Thread.currentThread().getName();
    while (true) {
      Integer value = set.pollFirst();
      if (value == null) continue;
      System.out.printf(threadName + ": removed%5d%n", value);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
      if (Thread.interrupted()) break;                                     // (
    }
  };

  public static void main(String[] args) {
    // Create and populate the set:                                           (
    set = new ConcurrentSkipListSet<>();
    new Random().ints(10, 0, 1000).forEach(val -> set.add(val));
    System.out.println(set);

    // Create an executor service to execute two tasks:                       (
    ExecutorService exs = Executors.newFixedThreadPool(2);
    try {
      exs.submit(sumValues);
      exs.submit(removeValues);

      ConcUtil.snooze(5, TimeUnit.SECONDS);                                // (
    } finally {
      System.out.println("Shutting down now.");
      exs.shutdownNow();                                                   // (
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
[20, 100, 236, 263, 299, 359, 548, 552, 591, 686]
pool-1-thread-1: sum     3654
pool-1-thread-2: removed   20
pool-1-thread-1: sum     3634
pool-1-thread-2: removed  100
pool-1-thread-1: sum     3534
pool-1-thread-2: removed  236
Shutting down now.
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

Concurrent Maps

Figure 23.5 shows the inheritance hierarchy of concurrent maps in the `java.util .concurrent package`. Note the interfaces `ConcurrentMap<K,V>` and `ConcurrentNavigableMap<K,V>` that extend the thread-unsafe map interfaces in the `java.util` package. Their implementations, `ConcurrentHashMap<K,V>` and `ConcurrentSkipListMap<K,V>`, provide efficient unsorted and sorted concurrent maps, respectively. The concrete map implementations are summarized in Table 23.9, together with their characteristics in Table 23.10.

Figure 23.5 *Concurrent Maps in the* `java.util.concurrent` *Package*

Table 23.9 *Concurrent Maps in the* `java.util.concurrent` *Package*

Concurrent maps	**Description**
Interface `ConcurrentMap<K,V>` extends `Map<K,V>`	A `Map` providing thread-safety and atomicity guarantees
`ConcurrentHashMap<K,V>` implements `ConcurrentMap<K,V>`	A hash table implementation supporting full concurrency of retrievals and high expected concurrency for updates
Interface `ConcurrentNavigableMap<K,V>` extends `ConcurrentMap<K,V>`, `NavigableMap<K,V>`	A `ConcurrentMap` supporting `NavigableMap` operations, and recursively so for its navigable sub-maps
`ConcurrentSkipListMap<K,V>` implements `ConcurrentNavigableMap<K,V>`	A scalable concurrent `ConcurrentNavigableMap` implementation that is highly efficient for traversal

Table 23.10 *Characteristics of Concurrent Maps*

Concurrent maps	**Duplicates**	**Ordering**	**Kind of iterator**	**Nulls**
`ConcurrentHashMap<K,V>` implements `ConcurrentMap<K,V>`	Unique keys	No order	Weakly consistent	Non-null keys or values
`ConcurrentSkipListMap<K,V>` implements `ConcurrentNavigableMap<K,V>`	Unique keys	Key sort order	Weakly consistent	Non-null keys or values

The `ConcurrentMap<K,V>` *Interface*

In order to maintain the guarantees of thread-safety and atomicity of operations, the `ConcurrentMap<K,V>` overrides the methods shown below from the `Map<K,V>` interface, and stipulates that concurrent implementations will honor these guarantees—as exemplified by the `ConcurrentHashMap<K,V>` class, affording efficient concurrent insertion and lookup. Details on these overridden methods can be found in the `Map<K,V>` interface (§15.8, p. 831).

The implementation of the `ConcurrentHashMap<K,V>` class conceptually divides the map into *segments* (also called *sub-maps*) that can be independently locked by a thread, thus allowing several threads to perform operations on the map concurrently.

Click here to view code image

```
interface ConcurrentMap<K,V> extends Map<K, V>  // §15.8, p. 831.

default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remapFunction)
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
default V computeIfPresent(K key,
               BiFunction<? super K,? super V,? extends V> remapFunction)

default void forEach(BiConsumer<? super K,? super V> action)

default V getOrDefault(Object key, V defaultValue)

default V merge(K key, V value,
                BiFunction<? super V,? super V,? extends V> remapFunction)
V putIfAbsent(K key, V value)
boolean remove(Object key, Object value)

V       replace(K key, V value)
boolean replace(K key, V oldValue, V newValue)
default void replaceAll(BiFunction<? super K,? super V,? extends V> func)
```

The `ConcurrentNavigableMap<K,V>` *Interface*

A concurrent, sorted map is defined by the `ConcurrentNavigableMap<K,V>` interface that extends both the `ConcurrentMap<K,V>` and the `NavigableMap<K,V>` interfaces. It overrides the methods shown below from its superinterfaces. Details on these overridden methods can be found in the `ConcurrentMap<K,V>` interface (p. 1491) and in the `NavigableMap<K,V>` interface (§15.10, p. 845).

The `ConcurrentSkipListMap<K,V>` class is an efficient implementation of the `Concurrent-NavigableMap<K,V>` class. This implementation is based on a concurrent variant of a *skip list* that is organized as a hierarchy of linked lists with subsequences of elements that are sorted, allowing efficient traversal, but at the extra cost of insertions in the skip list.

Click here to view code image

```
interface ConcurrentNavigableMap<K,V> extends ConcurrentMap<K,V>,
                                              NavigableMap<K,V>
NavigableSet<K> descendingKeySet()
ConcurrentNavigableMap<K,V> descendingMap()
```

Return a reverse-order `NavigableSet` view of the keys or a reverse-order `Concur-rentNavigableMap` of the entries contained in this map, respectively.

Click here to view code image

```
ConcurrentNavigableMap<K,V> headMap(K toKey)
ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)
```

Return a view of the portion of this map whose keys are strictly less than `toKey` or are less than or equal to `toKey`, if `inclusive` is `true` is specified.

```
NavigableSet<K> keySet()
NavigableSet<K> navigableKeySet()
```

Return a `NavigableSet` view of the keys contained in this map. These methods are equivalent.

Click here to view code image

```
ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)
ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                   K toKey, boolean toInclusive)
```

Return a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive, or from `fromKey` to `toKey`, where inclusion of the interval keys is explicitly specified.

Click here to view code image

```
ConcurrentNavigableMap<K,V> tailMap(K fromKey)
ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
```

Return a view of the portion of this map whose keys are greater than or equal to `fromKey`, or are greater than or equal to `fromKey`, if `inclusive` is `true` is specified.

Example 23.19 illustrates using the `ConcurrentHashMap<K,V>` class. A frequency map is created for rolling a dice at (3) and (4). A reader, declared as `diceResultsReader` at (1), repeatedly creates a new key set of the map and prints its contents. A remover, declared as `diceResultRemover` at (2), continuously removes the entry for a random dice result from the map. Both the reader and the remover take a snooze after printing the contents and removing an entry, respectively. All threads terminate on being interrupted, as the `shutdownNow()` method will initiate cancellation of all tasks at (6).

One reader and two removers are submitted to the service executor at (5). The output shows that the map contents reflect correctly that entries are removed and printed concurrently.

Example 23.19 *Using a Concurrent Map*

Click here to view code image

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;
import java.util.function.Function;
import java.util.stream.Collectors;

public class ConcurrentHashMapDemo {

  private static ConcurrentHashMap<Integer, Long> map;
  public static final int NUM_OF_THROWS = 1000;

  private static Runnable diceResultsReader = () -> {                     // (
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcurrentHashMap.KeySetView<Integer, Long> keySetView = map.keySet();
      String output = "";
      for (Integer key : keySetView) {
        Long value = map.get(key);
        output += " " + "<" + key + "," + value + ">";
      }
      System.out.println(threadName + ": {" + output + " }");
      ConcUtil.snooze(1000, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
    }
  };

  private static Runnable diceResultRemover = () -> {                     // (
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcUtil.snooze(500, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
      Integer key = ThreadLocalRandom.current().nextInt(1, 7);  // [1, 6]
      Long value = map.remove(key);
      if (value == null) continue;
      String removedEntry = threadName + ": removed "
                          + "<" + key + "," + value + ">";
      System.out.println(removedEntry);
    }
  };

  public static void main(String[] args) throws InterruptedException {
    map = new ConcurrentHashMap<>(6);                                     //
    new Random().ints(NUM_OF_THROWS, 1, 7)                                //
                .boxed()
                .parallel()
                .collect(Collectors.groupingByConcurrent(
                    Function.identity(),
                    () -> map,
                    Collectors.counting()));

    ExecutorService exs = Executors.newFixedThreadPool(3);
    try {
        exs.submit(diceResultsReader);                                      // (
        exs.submit(diceResultRemover);
        exs.submit(diceResultRemover);
        ConcUtil.snooze(5, TimeUnit.SECONDS);
      } finally {
        exs.shutdownNow();                                                  // (
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
pool-1-thread-1: { <1,160> <2,159> <3,170> <4,178> <5,158> <6,175> }
pool-1-thread-2: removed <1,160>
pool-1-thread-3: removed <6,175>
pool-1-thread-2: removed <5,158>
pool-1-thread-1: { <2,159> <3,170> <4,178> }
pool-1-thread-2: removed <4,178>
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-2: removed <2,159>
pool-1-thread-1: { <3,170> }
pool-1-thread-2: removed <3,170>
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
```

Blocking Queues

Queues (and deques) are the indisputable choice when choosing a collection to manage shared data in producer-consumer problems. The interfaces for thread-unsafe queues (`Queue`, `Deque`) in the `java.util` package have been enhanced in the `java.util.concurrent` package to provide a wide variety of blocking queues that are thread-safe (Figure 23.6). These thread-safe queues are *blocking* because a thread *blocks* when trying to add an element and the queue is full, or when trying to remove an element and the queue is empty.

Figure 23.6 *Blocking Queues in the* `java.util.concurrent` *Package*

A summary of interfaces and classes that implement blocking queues and deques provided by the `java.util.concurrent` package is given in Table 23.11. Salient characteristics of these queues and deques are summarized in Table 23.12. None of the queues or deques allow a `null` value for an element, but they do allow duplicates. Being queues, FIFO ordering is the norm and a majority of them are weakly consistent when it comes to serial access.

Table 23.11 *Blocking Queues in the* `java.util.concurrent` *Package*

Blocking queues	**Description**
Interface `BlockingQueue<E> extends Queue<E>` (Table 23.13)	A `Queue` that additionally supports operations that wait for the

	queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
`LinkedBlockingQueue<E> implements BlockingQueue<E>`	An optionally bounded blocking queue based on linked nodes.
`ArrayBlockingQueue<E> implements BlockingQueue<E>`	A bounded blocking queue backed by an array. A fairness policy can be specified so that threads blocked for insertion and removal are processed in FIFO order.
`PriorityBlockingQueue<E> implements BlockingQueue<E>`	An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking retrieval operations. Implementation is based on a binary heap.
`SynchronousQueue<E> implements BlockingQueue<E>`	A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. No elements are stored.
`DelayQueue<E extends Delayed> implements BlockingQueue<E>`	An unbounded blocking queue of `Delayed` elements, in which an element can only be taken when its delay has expired. Elements implement the `java.util.concurrent.Delay` interface. Whether an element is available or not, it still counts toward the size of the queue. Implementation is based on a `PriorityQueue` instance that in turn uses a binary heap.
Interface `TransferQueue<E> extends BlockingQueue<E>`	A `BlockingQueue` in which producers may wait for consumers to receive elements.
`LinkedTransferQueue<E> implements TransferQueue<E>`	An unbounded blocking `TransferQueue` based on linked nodes. The elements in the queue are ordered in FIFO order with respect to a given producer waiting for consumers.
Interface `BlockingDeque<E> extends BlockingQueue<E>, Deque<E>` (Table 23.14)	A `Deque` that additionally supports blocking operations that wait for the deque to become nonempty when retrieving an element, and wait for space to become available in the deque when storing an element.
`LinkedBlockingDeque<E> implements BlockingDeque<E>`	An optionally bounded blocking deque based on doubly linked nodes.

Table 23.12 *Characteristics of Blocking Queues*

Blocking queue	Ordering	Duplicates	Null values	Iterator
`LinkedBlockingQueue<E> implements BlockingQueue<E>`	FIFO	Allowed	Not allowed	Weakly consistent
`ArrayBlockingQueue<E> implements BlockingQueue<E>`	FIFO	Allowed	Not allowed	Weakly consistent
`PriorityBlockingQueue<E> implements BlockingQueue<E>`	Natural ordering	Allowed	Not allowed	Fail-fast
`SynchronousQueue<E> implements BlockingQueue<E>`	Not applicable	Not applicable	Not applicable	Not applicable
`DelayQueue<E extends Delayed> implements BlockingQueue<E>`	Longest-delayed order	Allowed	Not allowed	Fail-fast
`LinkedTransferQueue<E> implements TranferQueue<E>`	FIFO	Allowed	Not allowed	Weakly consistent
`LinkedBlockingDeque<E> implements BlockingDeque<E>`	FIFO/LIFO	Allowed	Not allowed	Weakly consistent

The `BlockingQueue<E>` *Interface*

The `java.util.concurrent.BlockingQueue` interface extends the `java.util.Queue` interface, as shown in Figure 23.6. Compared to the operations shown for the `Queue` interface in Table 23.7, the `BlockingQueue` interface provides new insert and remove operations that can block or can time out—shown in the two rightmost columns in Table 23.13. Classes that implement the `BlockingQueue` interface are shown in Table 23.11, where general-purpose queues are listed at the top, down to more specialized thread-safe queues. The `LinkedBlockingQueue` class represents the all-round queue of choice in most cases.

Table 23.13 *Selected Methods in the* `BlockingQueue` *Interface*

[illegible] value				
[illegible] waits if necessary for specified time if no space				
[illegible] waits if necessary for specified time for head element, and returns `null` *if no element becomes available*				
[illegible] [illegible] `NoSuchElement-Exception`				

The `TransferQueue<E>` *Interface*

The `TransferQueue` interface extends the `BlockingQueue` interface to provide queues where producers wait for consumers to receive elements (Figure 23.6). The interface defines additional methods to utilize this property.

Click here to view code image

```
void transfer(E e)
boolean tryTransfer(E e)
boolean tryTransfer(E e, long timeout, TimeUnit unit)
```

The first method transfers the element to a consumer, waiting if necessary to do so.

The second method transfers the element to a waiting consumer immediately, if possible. It returns `true` if the element was transferred; otherwise, it returns `false`.

The third method transfers the element to a consumer if it is possible to do so before the timeout elapses. It returns `true` if successful, and `false` if the specified waiting time elapses before completion, in which case the element is not left enqueued.

Click here to view code image

```
boolean hasWaitingConsumer()
int getWaitingConsumerCount()
```

> The first method returns `true` if there is at least one consumer waiting to receive an element via `BlockingQueue.take()` or a timed poll.
> The second method returns an estimate of the number of consumers waiting to receive elements via `BlockingQueue.take()` or a timed poll.

The `BlockingDeque<E>` *Interface*

The `java.util.concurrent.BlockingDeque` interface extends both the `java.util.Deque` interface and the `BlockingQueue` interface, as shown in Figure 23.6. Compared to the operations shown for the `ConcurrentLinkedDeque` interface in Table 23.8, the `Blocking-Deque` interface provides new insert and remove operations, both at the head and tail of a deque, that can block or can time out—shown in the light-gray rows in Table 23.14. Methods inherited from the `BlockingQueue` interface are marked with an asterisk (*) in Table 23.14. The `LinkedBlockingDeque` class that implements the `BlockingDeque` interface is shown in Table 23.11. This class represents an all-round implementation of a blocking deque.

Table 23.14 *Selected Methods in the* `BlockingDeque` *Interface*

Insert at the head	**Insert at the tail**	**Runtime behavior on failure**
`offerFirst(e)`	`offerLast(e)`, `offer(e)*`	*Returns* `false` *if full*
`addFirst(e)`	`addLast(e)`, `add(e)*`	*Throws* `IllegalStateException`
`putFirst(e)`	`putLast(e)`, `put(e)*`	*Blocks if full*
`offerFirst(e, time, unit)`	`offerLast(e, time, unit), offer(e, time,`	*Times out*
Remove from the head	**Remove from the tail**	**Runtime behavior on failure**
`pollFirst()`	`pollLast()`, `poll()*`	*Returns* `null` *if empty*
`removeFirst()`	`removeLast()`, `remove()*`	*Throws* `NoSuchElementException`
`takeFirst()`	`takeLast()`, `take(`	*Blocks if empty*
`pollFirst(time, unit)`	`pollLast(time, unit)`	*Times out*
Examine at the head	**Examine at the tail**	**Runtime behavior on failure**
`peekFirst()`	`peekLast()`, `peek()*`	*Returns* `null` *if empty*
`getFirst()`	`getLast()`, `element()*`	*Throws* `NoSuchElementException`

Example 23.20 demonstrates using the `LinkedBlockingQueue` class that implements the `BlockingQueue` interface. A producer is defined at (2) that puts a fixed number of random values (between 0 and 100) in the blocking queue. Note that the `put()` call at (3) can block. The thread sleeps for a little while after each `put()` operation. When done, it puts a *poison value* (which is not a legal value) in the queue at (4), that is interpreted as no more values are in the queue.

A consumer is defined at (5) that continuously takes a value from the blocking queue at (7) and prints it. It sleeps for a little while after each `take()` operation that can block. Before taking a value, it checks at (6) to see if the value at the head of the queue is the poison value. If so, the infinite loop is terminated, and thereby the consumer as well.

The `LinkedBlockingQueue` class is instantiated at (8). Note that the blocking queue is bounded, but the number of values to put in the queue is greater than its capacity—which can result in the `put()` operation to block if the queue is full. A service executor is created at (9). One producer and two consumers are submitted to the service executor at (10). The call to the `shutdown()` method at (11) allows threads to complete execution.

The output from the program shows that the producer puts the values in the blocking queue and the two consumers take the values and process them.

Example 23.20 *Linked Blocking Queue*

Click here to view code image

```
package concurrent;
import java.util.concurrent.*;

public class LinkedBlockingQueueDemo {

  public static final int UPPER_BOUND = 3;
  public static final int NUM_OF_VALUES = 5;
  public static final int STOP_VALUE = -1;

  private static BlockingQueue<Integer> queue;                            // (

  private static Runnable producer = () -> {                              // (
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    try {
      for (int i = 0; i < NUM_OF_VALUES; i++) {
        Integer value = tlrng.nextInt(100);
        queue.put(value);                                                 // (
        System.out.println(threadName + ": put " + value);
        Thread.sleep(tlrng.nextInt(200));
      }
      queue.put(STOP_VALUE);                                              // (
      System.out.println(threadName + ": done.");
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
  };

  private static Runnable consumer = () -> {                              // (
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    while (true) {
      try {
        Integer head = queue.peek();                                      // (
        if (head != null && head.equals(STOP_VALUE)) {
            System.out.println(threadName + ": done.");
            break;
        }
        Integer value = queue.take();                                     // (
        System.out.println(threadName + ": processing " + value);
        Thread.sleep(tlrng.nextInt(1000));
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  };

  public static void main(String[] args) {
    queue = new LinkedBlockingQueue<>(UPPER_BOUND);                       // (
    ExecutorService exs = Executors.newFixedThreadPool(3);                // (
    try {
      exs.submit(producer);                                               // (
      exs.submit(consumer);
      exs.submit(consumer);
    } finally {
      exs.shutdown();                                                     // (
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
pool-1-thread-1: put 43
pool-1-thread-2: processing 43
pool-1-thread-1: put 84
pool-1-thread-3: processing 84
pool-1-thread-1: put 60
pool-1-thread-3: processing 60
pool-1-thread-1: put 55
pool-1-thread-1: put 17
pool-1-thread-3: processing 55
pool-1-thread-1: done.
pool-1-thread-3: processing 17
pool-1-thread-2: done.
pool-1-thread-3: done.
```

Copy-on-Write Collections

The copy-on-write collections comprise special-purpose concurrent lists and sets that are recommended when read operations vastly outnumber mutative operations on the collection. The classes `CopyOnWriteArrayList` and `CopyOnWriteArraySet` implement the `List` and the `Set` interfaces in the `java.util` package (Figure 23.7). A summary of the copy-on-write classes in the `java.util.concurrent` package is given in Table 23.15. Salient characteristics of these collections are summarized in Table 23.16.

Figure 23.7 *Copy-on-Write Collections in the* `java.util.concurrent`

Package

Table 23.15 *Copy-on-Write Collections in the* `java.util.concurrent` *Package*

Copy-on-write collections	Description
`CopyOnWriteArrayList<E> implements List<E>`	A thread-safe variant of `ArrayList` in which all mutative operations (add, set, etc.) are implemented by making a fresh copy of the underlying array.
`CopyOnWriteArraySet<E> implements Set<E>`	A `Set` that uses an internal `CopyOnWriteArrayList` for all of its operations.

Table 23.16 *Characteristics of Copy-on-Write Collections*

Copy-on-write collections	Duplicates	Ordered	Kind of iterator
`CopyOnWriteArrayList<E> implements List<E>`	Yes	Insertion order	Snapshot-style
`CopyOnWriteArraySet<E> implements Set<E>`	No	No	Snapshot-style

The `CopyOnWriteArrayList<E>` *Class*

The `CopyOnWriteArrayList` class implements the `java.util.List` interface (§15.3, p. 801), providing a thread-safe list that is recommended for a vast number of concurrent read and traversal operations. It uses an underlying array to implement its operations. By default, such a list is immutable, guaranteeing thread-safety of concurrent operations. Any mutative operation (add, set, remove, etc.) is done on a new *copy* of its entire underlying array which then supersedes the previous version for subsequent operations. For efficiency reasons, the list size should be kept small and modifications should be minimized, as they are expensive, incurring the copying cost of its underlying array and extra space.

Traversal via snapshot-style iterators is efficient and thread safe—there is no need for any external synchronization, since it relies on the immutable state of the underlying array at the time the iterator is constructed (Example 23.21). Note that immutability of the list during traversal rules out the `remove()` operation of the iterator, as this can invalidate its snapshot traversal guarantee if allowed. In the code below, the `Iterator.remove()` method at (2a) results in an `UnsupportedOperation-Exception`, whereas the `List.remove()` method is allowed since it will be performed on a new copy of the underlying array. Note that a thread-unsafe `ArrayList` instance would allow both (2a) and (2b).

Click here to view code image

```
List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (1a)
cowlist.addAll(Arrays.asList(10, 20, 30));
Iterator<Integer> iter = cowlist.iterator();
```

```
  while (iter.hasNext()) {
    Integer i = iter.next();
    if (i == 20) {
//    iter.remove();                       // (2a) UnsupportedOperationException
      cowlist.remove(i);                   // (2b) OK

      continue;
    }
    System.out.print(i + " ");             // With (2b): 10 30
  }
```

In addition to the methods of the `List` interface, the following useful methods are also defined by the `CopyOnWriteArrayList` class:

> `boolean addIfAbsent(E e)`
>
> Appends the element, if not present.
>
> Click here to view code image
>
> `int addAllAbsent(Collection<? extends E> c)`
>
> Appends to the end of this list all of the elements in the specified collection that are not already contained in this list.

The `CopyOnWriteArraySet<E>` *Class*

The `CopyOnWriteArraySet` class implements the `java.util.Set` interface (§15.4, p. 804). It does not implement any additional methods. Internally it uses a `CopyOnWriteArrayList`, and therefore shares the same basic properties with the list, except that, being a set, it does not allow duplicates and its elements have no ordering.

Example 23.21 illustrates the snapshot-style iterator of a `CopyOnWriteArrayList`. Such a list is created at (1) and populated with three values. A task to traverse the list and print its elements is defined by the `Runnable iter` at (2). Two threads are created at different times to execute this task at (4) and (6), respectively. The main thread modifies the list by adding a new value and removing a value after the start of the first thread. The output shows that the result from the first thread is not affected by the modifications done by the main thread, as its iterator only traverses those elements that were in the list when the iterator was created. The result from the second thread shows the state of the list when it was created—that is, after the list was modified by the main thread.

Example 23.21 *Copy-on-Write Array List*

Click here to view code image

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class CopyOnWriteArrayListDemo {

  public static void main(String[] args) {
    List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (
    cowlist.addAll(Arrays.asList(1, 2, 3));

    Runnable iter = () -> {                                               // (
      String threadName = Thread.currentThread().getName();
      for (Integer i : cowlist) {
        System.out.println(threadName + ": " + i);
        ConcUtil.snooze(1, TimeUnit.SECONDS);
      }
    };

    // First iterator:
    new Thread(iter, "Iterator A").start();                               // (

    // Snooze, add, and remove in main thread.                            // (
    ConcUtil.snooze(1, TimeUnit.SECONDS);
    Integer newValue = 4;
    cowlist.add(newValue);
    System.out.println("New value added: " + newValue);
    Integer first = cowlist.remove(0);
    System.out.println("Value removed:   " + first);

    // Second iterator:
    new Thread(iter, "Iterator B").start();                               // (
  }
}
```

Probable output from the program:

```
Iterator A: 1
New value added: 4
Iterator A: 2
Value removed:   1
Iterator B: 2
Iterator A: 3
Iterator B: 3
Iterator B: 4
```

Review Questions

23.1 Which of the following statements is true?

Select the one correct answer.

a. A single thread executor service cannot be used to schedule tasks.

b. A fixed thread pool is configured with a given level of parallelism.

c. A scheduled executor service allows a task to be run with a specified delay.

d. A cached thread pool implements a work-stealing mechanism.

23.2 Which of the following statements is true?

Select the one correct answer.

a. The executor service’s `shutdown()` method cancels all tasks that are still running.

b. The executor service’s `shutdown()` method disables new tasks from being submitted.

c. The executor service’s `awaitTermination()` method cannot be interrupted.

d. The executor service’s `shutdownNow()` method awaits termination of tasks that are still running.

23.3 Which of the following statements is true about the `ReentrantReadWriteLock` class?

Select the one correct answer.

a. Its read lock prevents concurrent read operations from reading inconsistent data.

b. Its write lock prevents concurrent read operations from reading inconsistent data.

c. Its write lock acquisition cannot be interrupted.

d. Its read lock acquisition cannot be interrupted.

23.4 Given the following code:

Click here to view code image

```
var values = List.of("1", "2", "3", "4", "5", "6", "7", "8");
var list1 = new CopyOnWriteArrayList<Integer>();
for (String value: values) {
    new Thread(() -> {
        list1.add(Integer.valueOf(value));
    }).start();
}
System.out.println(list1);

var list2 = values.parallelStream().map(v->Integer.valueOf(v))
                                   .toList();
System.out.println(list2);

var list3 = new ArrayList<Integer>();
values.parallelStream().map(v->Integer.valueOf(v)).forEach(v->list3.add(
System.out.println(list3);
```

Which lists will always print the same values as in the initial list?

Select the one correct answer.

- **a.** `list1`
- **b.** `list2`
- **c.** `list3`
- **d.** `list1` and `list2`
- **e.** `list2` and `list3`
- **f.** `list1` and `list3`
- **g.** All of the lists
- **h.** None of the lists

23.5 Which of the following statements is true?

Select the one correct answer.

- **a.** Synchronized collections provide a synchronized `iterator()` method.
- **b.** Copy-on-write collections provide a synchronized `iterator()` method.
- **c.** Immutable collections provide a synchronized `iterator()` method.
- **d.** Synchronized collections require the programmer to implement a synchronized `iterator()` method.
- **e.** Copy-on-write collections require the programmer to implement a synchronized `iterator()` method.
- **f.** Immutable collections require the programmer to implement a synchronized `iterator()` method.

23.6 Which of the following statements is true about atomic variables?

Select the one correct answer.

- **a.** Atomic variables represent the smallest indivisible unit of data.
- **b.** Atomic variables ensure memory consistency using synchronized operations.
- **c.** Atomic variables provide methods that can throw an `InterruptedException`.
- **d.** Atomic variables do not utilize an intrinsic locking mechanism.

23.7 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
public class RQ7 {
  public static void main(String[] args) {
    AtomicLong aValue = new AtomicLong(0);
    Callable<Number> c = () -> {
      AtomicLong value = aValue;
        return value.incrementAndGet();
      };

    Collection<Callable<Number>> cc = List.of(c, c, c);
    ExecutorService es = Executors.newFixedThreadPool(2);
    List<Future<Number>> futures = null;
    try {
      futures = es.invokeAll(cc);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    es.shutdown();

    futures.stream().mapToLong(v-> {
    try {
      return v.get().longValue();
    }catch(Exception e) {
      return -1;
    }}).forEach(v->System.out.print(v));
    System.out.print(aValue);
  }
}
```

Which of the following output can be produced by this program? Select the two correct answers.

a. `1233`

b. `2133`

c. `1122`

d. `2122`

e. `2123`

23.8 Given the following code:

Click here to view code image

```
import java.util.concurrent.locks.*;
public class RQ8 {
  public static void main(String[] args) {

    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    Lock rl = rwl.readLock();
    Lock wl = rwl.writeLock();
    try {
      rl.lock();
      System.out.println("Read lock acquired");
      if (wl.tryLock()) {
        System.out.println("Write lock acquired");
      }
    } catch(Exception e) {
      System.out.println("Lock acquisition failed");
    } finally {
      rl.unlock();
      if (rwl.isWriteLocked()) {
        wl.unlock();
        System.out.println("Write lock released");
      }
    }
    System.out.println("The end");
    }
  }
```

What is the result?

Select the one correct answer.

a. `Read lock acquired The end`

b. `Read lock acquired Write lock acquired Write lock released The end`

c. `Read lock acquired Lock acquisition failed The end`

d. `Lock acquisition failed The end`

e. `Lock acquisition failed Write lock released The end`

23.9 Given the following code:

Click here to view code image

```
public class RQ9 {
  private static volatile int counter = 10;
  public static void main(String[] args) {
    Runnable r = () -> counter--;
    while (counter > 0) {
      new Thread(r).start();
      System.out.print(counter);
    }
  }
}
```

What is the result?

Select the one correct answer.

a. Either `9876543210` or `876543210-1` or 987654321

b. Either `9876543210` or `10987654321` or 987654321

c. `9876543210`

d. The result is unpredictable.

23.10 Given the following code:

Click here to view code image

```
import java.util.concurrent.*;
public class RQ10 {
  public static void main(String[] args) {
    try {
      ExecutorService es = Executors.newFixedThreadPool(2);
      Future<?> f1 = es.submit(() -> "acme");
      Future<?> f2 = es.submit(() -> {}); // (1)
      es.shutdown();                      // (2)
      Object o1 = f1.get();
      Object o2 = f2.get();               // (3)
      System.out.println(o1 + " " + o2);  // (4)
    } catch (InterruptedException | ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

What is the result?

Select the one correct answer.

a. An exception is thrown at (1).
b. An exception is thrown at (2).
c. An exception is thrown at (3).
d. An exception is thrown at (4).
e. `acme` null
f. `acme`

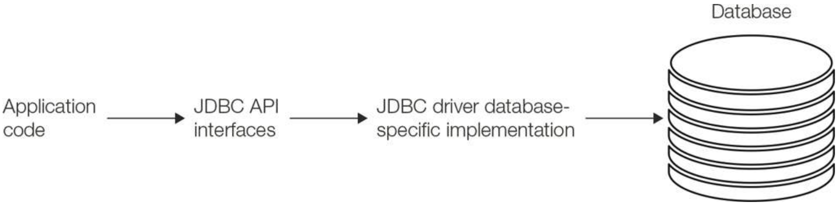
23.11 Given the following code:

Click here to view code image

```
import java.util.*
import java.util.concurrent.*;
import java.util.stream.*;

public class RQ11 {
  public static void main(String[] args) {
    Map<Integer, String> map =
        new ConcurrentHashMap<>(Map.of(1,"a",2,"b",3,"c",4,"d",5,"e"));
    List<Future<String>> results = new CopyOnWriteArrayList<>();

    ExecutorService es = Executors.newFixedThreadPool(3);
    for (int i = 1; i <= map.size(); i++) {
      final int key = i;
      Future<String> f = es.submit(() -> map.get(key).toUpperCase());
      if (i % 2 != 0) {
        f.cancel(true);
      }

      results.add(f);
    }
    es.shutdown();
    String result = (results.stream().allMatch(r -> r.isDone()))
        ? results.stream().filter(r -> !r.isCancelled()).map(r -> {
            try {
              return r.get();
            } catch (InterruptedException | ExecutionException e) {
              return "X";
            }
          }).collect(Collectors.joining())
        : "Z";
    System.out.println(result);
  }
}
```

Which of the following statements are true about the result of this program? Select the two correct answers.

a. The program terminates with an exception.
b. The program only prints the letters `B` or `D`.
c. The program never prints the letters `B` or `D`.
d. The program only prints the letters `A`, `C`, or `E`.
e. The program never prints the letters `A`, `C`, or `E`.
f. The program may print the letters `A`, `B`, `C`, `D`, or `E`.
g. The program may print the letter `X`.
h. The program may print the letter `Z`.



Database Connectivity 24

Chapter Topics

- Understanding the basics of relational databases: relational tables and SQL
- Understanding the role of the JDBC driver, and establishing a database connection using such a driver
- Understanding the handling of JDBC resources, and closing them in a responsible way
- Writing code to create and execute SQL statements: basic statement, prepared statement, and callable statement in the JDBC API
- Writing code to process query results, including customizing the result set returned by a query
- Discovering database capabilities and reading result set metadata
- Writing code to implement transaction control

Java SE 17 Developer Exam O	
[11.1] Implement localization using locales, resource bundles, parse and format messages, dates, times, and numbers including currency and percentage values	*§24.1, p. 1512,* *to* *§24.8, p. 1545*

Java SE 11 Developer Exam O	
[11.1] Connect to and perform database SQL operations, process	*§24.1, p. 1512,*

to query results using JDBC API
§24.8, p. 1545.

Database is a general term that describes organized data storage, controlled by a *Database Management System* (DBMS). There are many different such systems from different providers and vendors, both commercial and open source.

The responsibilities of a DBMS typically include tasks such as secure and control data access, keep data safe and consistent, and query the data. Databases can be classified by the way in which they store information, such as hierarchical, network, object, and relational, each with their own advantages and disadvantages. Many modern DBMS present a mix of different database features and capabilities.

This chapter discusses how to develop Java applications that use a relational database, which is one of the most commonly used database models. The Java Database Connectivity (JDBC) specification allows development of Java applications that can leverage a relational database independent of a specific DBMS.

The coverage of JDBC in this chapter is in no way exhaustive. The topics covered provide the basic essentials to write Java applications that use the JDBC API to interact with a relational database in an independent way. Prior knowledge of databases is advantageous, but not essential, as the basics are covered in this chapter. However, to be proficient, the large body of literature and software in this area should be consulted.

24.1 Introduction to Relational Databases

Relational databases are based on the *relational model* that provides a standardized way to represent and query data. The data is stored in *tables* and queried using a *query language*.

Relational Tables

Relational databases store information in *tables*. Each table is designed to represent a business entity that applications need to store information about. Each table is composed of *columns* (a.k.a. *attributes*). Each column represents specific information that describes a specific characteristic of a business entity that the table represents.

Each column has a *name* and describes information of a specific *type*, such as fixed-or variable-length character data (type `CHAR` or `VARCHAR`), numeric values (type `NUMBER` or `INTEGER`), date and time values (type `DATE` or `TIMESTAMP`), and many other types. Different database providers may support different data types.

Each table may be defined with a number of *constraints* that enforce business rules associated with this entity. For example, constraints may be used to enforce uniqueness of column values (`PRIMARY KEY`), or that these values must always be present (`NOT NULL`), or that they are restricted to a specific list or range of values. Different database providers may support different constraint types.

Actual business data is stored in tables as *rows* (a.k.a. *tuples*). Each row is composed of values that follow the exact structure of columns defined by a given table.

As an example, a music catalogue application needs to store information about music compositions in a database. Let us assume that each composition has the following attributes:

- An International Standard Recording Code (*ISRC*), which uniquely identifies a composition
- A title
- A duration

Thus to store this information in a relational database would require a `compositions` table with `isrc`, `title`, and `duration` columns. Assume that all of these are mandatory pieces of information for every composition object, and that `isrc` and `title` are both of type `VARCHAR`, and `duration` is of type `INT`. The definition of the `compositions` table is shown in Table 24.1.

Table 24.1 *Definition of the* `compositions` *Table*

Column	type	constraints
`isrc`	`VARCHAR(12)`	`PRIMARY KEY`
`title`	`VARCHAR(40)`	`NOT NULL`
`duration`	`INT`	`NOT NULL`

Note that column types often specify specific length or precision characteristics— for example, `VARCHAR(12)` in Table 24.1. Also, a `PRIMARY KEY` constraint is usually implicitly mandatory `NOT NULL` (does not allow `null` values). In Table 24.1, the `isrc` column name uniquely identifies a composition, hence its constraint is `PRIMARY KEY` and implicitly `NOT NULL`. The other two column names `title` and `duration` are mandatory for each composition, hence their constraint is also `NOT NULL`.

Table 24.2 shows example data (four rows) stored in the `compositions` table, which satisfies the *table definition* given in Table 24.1. Each row in Table 24.2 represents a unique music composition.

A table definition can be loosely compared to a class that describes a number of attributes, and a row can be compared to an object (instance) of that specific class. A spreadsheet is another example of organizing data in rows and columns.

Table 24.2 *Data in the* `compositions` *Table*

cid	**title**	**duration**
ushm91736697	Vacation	231
ushm91736698	Rage	308
ushm91736699	Why Don't	178
ushm91736700	Something Happened	147

Basic SQL Statements

Structured Query Language (SQL) is used to perform relational database operations. Despite the existence of an ANSI (American National Standards Institute) SQL standard, different database providers may provide SQL implementations that are not exactly the same and may contain nuanced differences and proprietary additions.

SQL statements can be logically grouped into a number of languages, in which they are identified by SQL keywords:

- *Data Manipulation Language* (*DML*) comprises the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements, used to query, create, modify, and remove table rows. Sometimes the `SELECT` statement is described to be in its own *Data Query Language* (*DQL*) group.
- *Data Definition Language* (*DDL*) comprises the `CREATE`, `ALTER`, and `DROP` statements, whose purpose is to create, modify, and remove tables. We will refer to the `CREATE`, `ALTER`, and `DROP` statements as *DDL operations*.
- *Transaction Control Language* (*TCL*) comprises the `COMMIT` and `ROLLBACK` statements, whose purpose is to save or undo pending changes, respectively.

Often the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements (in that order) are referred to as *CRUD operations*, denoting the *Create*, *Read/Retrieve*, *Update*, and *Delete* operations implied by the acronym, respectively. Note the name mismatch of the CRUD operations and the SQL statements is implied. In particular, the set of CRUD operations does *not* include the `CREATE` statement in SQL that creates relational tables.

The rest of this subsection presents basic syntax patterns and examples of creating relational tables and the CRUD operations in SQL. Note that the semicolon (`;`) is a *statement terminator* in SQL.

The `CREATE` *Statement*

The `CREATE` statement can be used to create a new table in the database:

Click here to view code image

```
CREATE TABLE table_name (column_definitions);
```

where *table_name* is the name of the table to be created, and *column_definitions* is a comma-delimited list that describes each column in this table:

Click here to view code image

```
column_name column_type constraints, ...
```

The following statement creates an empty `compositions` table:

Click here to view code image

```
CREATE TABLE compositions (isrc VARCHAR(12) PRIMARY KEY,
                           title VARCHAR(40) NOT NULL, duration INT NOT NULL);
```

The `CREATE` statement above will create an empty relational table whose columns will correspond to the table shown in Table 24.2.

The `INSERT` *Statement*

The `INSERT` statement can be used to insert a new row into a table:

Click here to view code image

```
INSERT INTO table_name VALUES (actual_values);
```

where *actual_values* in the `VALUES` clause is a comma-delimited list that provides values for the columns of the table specified by *table_name*.

The following statement inserts a new row into the `compositions` table:

Click here to view code image

```
INSERT INTO compositions VALUES ('ushm91736697', 'Vacation', 231);
```

The number of rows in the `compositions` table increases by 1. By default, the values specified are interpreted in the same order as the columns are defined in the table. Note that string literals in SQL are enclosed in single quotes (`'`).

The `SELECT` *Statement*

The `SELECT` statement can be used to query or read rows from the database:

Click here to view code image

```
SELECT column_list FROM table_name WHERE row_filter;
```

The *column_list* is a comma-delimited list of column names, whose values are selected from the table specified with *table_name* in the `FROM` clause. The `'*'` symbol can be used as *column_list* to specify that all columns should be included in the returned result. A list of conditions is specified by *row_filter* in the `WHERE` clause to determine which specific rows should be selected. Such conditions may be constructed using a variety of operators to compare column values, such as `=` (*equals*), `>` (*greater than*), `<` (*less than*), `LIKE` (which can use `"%"` wildcards), and many others. Conditions in *row_filter* can be combined using `AND` and `OR` operators.

Click here to view code image

```
column_name = some_value AND column_name LIKE some_value
```

The following statement selects values of `isrc` and `title` for all rows in the `compositions` table whose `duration` is greater than `200` and whose `title` starts with `'V'`:

Click here to view code image

```
SELECT isrc, title FROM compositions WHERE duration > 200 AND title LIKE 'V%';
```

Applied to Table 24.2, the above query will return the values of `isrc` and `title` in the first row, which is the only row that meets the *row_filter* criteria—that is, `(ushm91736697, Vacation)`.

The `WHERE` clause is optional. In that case, the values of *column_list* in the entire table are returned.

```
SELECT * FROM compositions;
```

The above `SELECT` statement will return all the rows in the specified table.

The `UPDATE` *Statement*

The `UPDATE` statement can be used to update specific rows in a table. In other words, this statement modifies zero or more rows in the table:

Click here to view code image

```
UPDATE table_name SET column_name_value_pairs WHERE row_filter;
```

where *column_name_value_pairs* in the `SET` clause is a comma-delimited list of pairs of column names and values:

Click here to view code image

```
column_name = column_value, ...
```

and *row_filter* in the `WHERE` clause specifies a list of conditions to indicate which specific rows should be updated.

All rows that meet the *row_filter* criteria in *table_name* will have their columns specified in the *column_name_value_pairs* set to the corresponding values in *column_name_value_pairs*.

The following statement updates a `duration` value for an existing row in the `compositions` table:

Click here to view code image

```
UPDATE compositions SET duration = 240 WHERE isrc LIKE '%91736697'
                                          OR duration > 231;
```

The above `UPDATE` statement will update the first row `(isrc '91736697')` in Table 24.2 by setting the `duration` to `240`.

The `WHERE` clause is optional. When omitted, all rows will have their columns specified in the *`column_name_value_pairs`* set to the corresponding values in *`column_ name_value_pairs`*.

Click here to view code image

```
UPDATE compositions SET duration = 250;
```

The above `UPDATE` statement will update the `duration` column for all rows to `250` in Table 24.2.

The `DELETE` *Statement*

The `DELETE` statement can be used to delete specific rows in a table that meet the criteria in the `WHERE` clause. This statement deletes zero or more rows from the specified table:

Click here to view code image

```
DELETE FROM table_name WHERE row_filter;
```

where *row_filter* in the `WHERE` clause is the criteria for which rows should be deleted from *table_name* in the `FROM` clause.

The following statement deletes a row from the `compositions` table:

Click here to view code image

```
DELETE FROM compositions WHERE isrc = 'ushm91736697';
```

The row with `isrc` having the value `'ushm91736697'` in the `compositions` table will be deleted from Table 24.2.

The `WHERE` clause is optional. Care must be taken, as omitting it will delete all rows from the table.

```
DELETE FROM compositions;
```

There are many other statements available in SQL described by the ANSI SQL standard and additional statements which may be implemented by database providers.

This book does not have a goal of covering SQL in detail, but provides a brief introduction to ensure a minimum level of understanding of how relational databases operate and thereby an understanding of how Java programs interact with relational databases.

Java certification focuses on Java, not on SQL. However, it is essential to learn database concepts and understand SQL if one wants to work with relational databases.

24.2 Introduction to JDBC

The *Java Database Connectivity* (JDBC) protocol provides database interaction capabilities for Java programs. The JDBC specification is not database provider specific, allowing Java programs to interact with any database.

The *JDBC API* is defined in the `java.sql` package. It comprises a number of interfaces that Java programs use to represent database connections, SQL statements, query results, and much more.

The JDBC API is implemented by a number of database-specific *JDBC drivers*. The job of a JDBC driver is to provide database-specific, native protocol implementation of the JDBC API. For example, Oracle, MySQL, DerbyDB, and others provide JDBC drivers that all implement the same JDBC API, but in a different database-specific manner.

A Java application can dynamically load the JDBC driver as required and connect to different databases, provided that the JDBC driver is present in the class path or module path of the application.

Figure 24.1 illustrates the layered approach to database connectivity. Java developers write code that utilizes JDBC interfaces defined by the `java.sql` package. These interfaces are implemented by different JDBC drivers. Java developers should only write code that utilizes JDBC API interfaces and not use JDBC drivers directly, so that software portability is not compromised. This allows Java applications to maintain database-provider neutrality, and potentially switch JDBC drivers and database providers without modifying application code. This approach is designed to decouple the application code from specific database providers, potentially allowing the application to switch between different databases and even use different databases at the same time.

Figure 24.1 *Layered Database Connectivity*

In order to connect to a database, the Java application has to perform the following tasks:

1. Ensure that the relevant JDBC driver is available in the class path or module path of the application.
2. Load the JDBC driver to memory.
3. Establish the database connection.

Once a database connection is established, a typical interaction scenario with the database proceeds as follows:

1. *Create* SQL statements.
2. *Execute* SQL statements.
3. *Process* query results.
4. *Close* the JDBC resources.

Closing JDBC Resources

It is important to ensure that all JDBC resources are properly closed once they are no longer needed.

All JDBC API methods can throw a `java.sql.SQLException`. A `SQLException` is a checked exception, which in addition to the usual Java error message, also wraps up database error information such as the SQL state and an error code.

The skeletal code below will always catch the `SQLException` and handle it:

Click here to view code image

```
try {
/* execute JDBC operations */
} catch (SQLException e) {
  String state = e.getSQLState();
  int code = e.getErrorCode();
}
```

The logic of the JDBC application can be generalized with the following skeletal code:

Click here to view code image

```
try {
  /* establish database connection     */
  /* create and execute SQL statements */
  /* process results                   */
} catch (SQLException e) {
  /* handle any errors */
} finally {
  /* close result sets */
  /* close statements  */
  /* close connection  */
}
```

It is important to remember that the closing order of these objects is significant: First close any result set objects, then close statements, and then close connections. No Java exceptions are actually produced if you try to close in a different order. However, this would prevent resources from being promptly released and thus can result in memory leaks.

JDBC interfaces that represent connection, statement, and result set objects all implement the `AutoCloseable` interface. Therefore, they can be used in the `try`-with-resources construct, in which case they are automatically closed in the implicit `finally` block inserted by the compiler, as shown below.

Click here to view code image

```
try (/* establish database connection     */
     /* create and execute SQL statements */)
{
  /* process results */
} catch (SQLException e) {
  /* handle any exceptions */
} /* implicit finally block closes resources*/
```

24.3 Establishing a Database Connection

Interaction between the application and the database is in the context of a *connection*. Database connections are represented by the `java.sql.Connection` interface. JDBC drivers provide database-specific implementations of this interface.

The class `java.sql.DriverManager` provides the overloaded factory method `get-Connection()` that initializes and returns a database `Connection` object.

> Click here to view code image
>
> ```
> static Connection getConnection(String jdbcURL)
> static Connection getConnection(String jdbcURL, String username,
>                                 String password)
> static Connection getConnection(String jdbcURL, Properties info)
> ```

All versions of the `getConnection()` method require a JDBC URL and all throw a `SQLException`. Additional information may be required to establish a database connection, such as username and password, or other means of authentication such as digital signatures that can be set using the `Properties` object (§18.2, p. 1100). Note also that the `Connection` object returned is `AutoCloseable`, and therefore it is best handled in a `try`-with-resources construct to ensure that the connection is closed when done.

JDBC URL

The general syntax of the *JDBC URL* is as follows:

Click here to view code image

```
protocol:provider:driver_type:database_specific_connection_details
```

The *protocol* is always specified as `"jdbc"`. Other details may vary depending on the database provider. For example, to connect to the

Apache Derby database you need to specify the *provider* as `"derby"`, followed by the description of where this database is located, which would include *host*, *port*, and *database name* information.

Click here to view code image

```
jdbc:derby:localhost:1521:musicDB
```

The example below shows a more sophisticated JDBC driver that may provide different connectivity mechanisms, each identified by an appropriate *subprotocol*. In this case, the JDBC URL example shows how one can specify an Oracle database connection using the most frequently used *thin* protocol implementation that provides enhanced security features.

Click here to view code image

```
jdbc:oracle:thin:@localhost:1521:musicDB
```

Not all JDBC drivers provide alternative connectivity methods. In the Oracle JDBC driver case, alternative connectivity mechanisms, such as *thin*, *OCI*, or *kprb*, can be used. This could be the case with many other providers of JDBC drivers. In the case of a Derby database, it can be accessed as a separate process or be embedded inside a Java program, which would make the JDBC URL look different. It is best to refer to JDBC driver-specific documentation to identify available choices and the rationale behind selecting one or the other connectivity mechanism.

Getting a Database Connection

To create a connection to a Derby database, the following code can be emulated:

Click here to view code image

```
String jdbcURL  = "jdbc:derby:localhost:1521:musicDB";
String username = "joe";
String password = "welcome1";
try (Connection connection = DriverManager.getConnection(jdbcURL,
                                                        username, password))
  /* use the connection. */
} catch (SQLException e) {
  e.printStackTrace();
}
```

A portion of the URL that indicates a driver instructs the `DriverManager` to load the appropriate driver implementation to memory. This of course would fail if this JDBC driver is not found in the class path or module path.

Connecting to the `musicDB` *Database*

We will use the `musicDB` database which consists of the `compositions` table shown in Table 24.2. Instructions for downloading the files for the `musicDB` database and the Derby JAR files can be found on this book's website. The structure of the working directory is assumed to be the following:

Click here to view code image

```
Working directory
 ?
 ??? db-derby-10.15.2.0-lib/   <== Directory with the Apache Derby distribution
 ?   ?
 ?   ??? lib/                  <== Directory with the Derby JAR files
 ?
 ??? musicDB/                  <== Directory with the musicDB database files
 ?
 ??? dbDemo/                   <== Package with examples used in this chapter
 ?   ?
 ?   ??? JDBCConnection.java
 ?   ?
... ...
```

Example 24.1 shows how to obtain a connection to the `musicDB` database. Note that in the JDBC URL, the relative path of the `musicDB` directory in the working directory is `musicDB`. The class path must also be set to access the Derby JAR files. In the code below, the implicit `finally` clause of the `try`-with-resources statement guarantees to close the connection when done.

Example 24.1 *Connecting to the* `musicDB` *Database*

Click here to view code image

```
package dbDemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCConnection {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (Connection connection = DriverManager.getConnection(jdbcURL)) {
      /* use the connection. */
      System.out.println(connection);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
org.apache.derby.impl.jdbc.EmbedConnection@1201454821 (XID = 1301),
(SESSIONID = 1), (DATABASE = musicDB), (DRDAID = null)
```

Legacy JDBC Driver Management

Prior to JDBC version 4.0, drivers had to be loaded into memory explicitly, before using them to establish database connections. There are several ways in which this could be achieved:

- Instantiate and register the database-specific JDBC driver class using the `java.sql.DriverManager.registerDriver()` method:

  Click here to view code image

  ```
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  ```

- Load a database-specific JDBC driver class using the `java.lang.Class.forName()` method:

  Click here to view code image

  ```
  try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
  } catch (ClassNotfoundException c) { }
  ```

- Load a database-specific JDBC driver class using the command-line `-D` option:

  Click here to view code image

  ```
  java -Djdbc.drivers=oracle.jdbc.driver.OracleDriver ...
  ```

From JDBC 4.0 onward, JDBC drivers are automatically loaded, and no explicit driver loading or registration is required. However, old-style code would still function for backward compatibility reasons.

24.4 Creating and Executing SQL Statements

The JDBC API defines three interfaces that represent SQL *statements*:

- `java.sql.Statement` represents a basic statement (p. 1523).

This type of statement can be used to execute any SQL operation, such as CRUD or DDL operations. However, it is associated with performance and security issues. Thus it is not generally recommended, except for very simple operations that do not require dynamic parameterization and that are not reused within a program.

- `java.sql.PreparedStatement` is a subinterface of the `Statement` interface, that represents a statement capable of *substitutional parameterization* (p. 1526). This type of statement has the same capabilities as a basic statement, but is considered to be better from both performance and security perspectives, and thus is a recommended choice for representing SQL operations in general.
- `java.sql.CallableStatement` is a subinterface of the `PreparedStatement` interface, that represents a statement that invokes a *stored procedure* or *function* (p. 1530) and is also capable of *substitutional parameterization*.

The result of executing a `SELECT` statement is a table of rows represented by the following interface:

- `java.sql.ResultSet` that represents a table whose contents can be read and processed

All statements are created using a previously initialized JDBC connection object. A statement is created using one of the methods of the `Connection` interface shown below. Depending on the method, a specific statement is returned. Note that the statement object returned is `AutoCloseable`, and should be used in a `try`-with-resources statement to ensure proper closing of recourses such objects represent.

The following methods are defined in the `Connection` interface:

> Click here to view code image

```
Statement         createStatement()
PreparedStatement prepareStatement(String sql)
CallableStatement prepareCall(String sql)
```

For overloaded versions of these methods in the `Connection` interface that can be used to customize the result set, see p. 1541.

Basic Statement

We first look at how to create and execute a basic statement. A `Statement` object should first be obtained from the `Connection` object. Note the declarations in the header of the `try`-with-resources statement that obtain a `Connection` object and create a `Statement` object, respectively. The `try`-with-resources statement will ensure that these resources are closed in the right order—that is, the reverse of their declarations.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) { // Obtain a Statement object.
  /* execute a query */
} catch (SQLException e) {
  e.printStackTrace();
}
```

Once a statement has been obtained, the following methods of the `Statement` interface can be used to execute and process a SQL query.

Click here to view code image

```
boolean execute(String sql)
```

The `execute()` method can be called to execute any operation, returning a `boolean` value which is `true` for the `SELECT` statement, `false` otherwise.

Click here to view code image

```
ResultSet executeQuery(String sql)
```

The `executeQuery()` method can only be called to execute `SELECT` statements, returning a `ResultSet` that is the result of executing the query. A `ResultSet` is also `AutoCloseable` and must be closed when done. Passing a non-`SELECT` operation will throw a `SQLException`.

Click here to view code image

```
int executeUpdate(String sql)
```

The `executeUpdate()` method can be called to execute any operations, except `SELECT` operations, returning an `int` value indicating the number of rows affected by the non-`SELECT` operation. Passing a `SELECT` operation will throw a `SQLException`.

> ```
> ResultSet getResultSet()
> ```
>
> Returns the `ResultSet` that was the result of executing a `SELECT` query using the current statement.
>
> ```
> int getUpdateCount()
> ```
>
> Returns the number of rows that were affected by the execution of a non-`SELECT` query using the current statement.

An appropriate execute method can be called depending on the type of SQL statement and on passing the SQL operation as a `String` parameter. The code below executes a `SELECT` query by calling the `execute()` method of the `Statement` interface, as shown at (1). This method returns a `boolean` value. The output from the code at (2) indicates that a `SELECT` query was processed.

Click here to view code image

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {      // Obtain a Statement object.
  String sql = "select * from compositions";    // SELECT query: select all ro
  boolean isSelectStmt = statement.execute(sql);          // (1) Execute the q
  System.out.println("SELECT statement? " + isSelectStmt);// (2) SELECT statem
                                                          //     true
} catch (SQLException e) {
  e.printStackTrace();
}
```

When calling the `execute()` method with a SQL statement as a `String` parameter, it might not be possible to know a priori what type of SQL statement will be executed. However, this method would return `true` if the statement happens to be a `SELECT` query; otherwise, it returns `false`. As shown in the code below, which analyzes the resulting value in an `if` statement, a `ResultSet` object can be retrieved or the number of rows affected by the statement, depending on the result.

Click here to view code image

```
boolean isSelectStmt = statement.execute(sql);
if (isSelectStmt) {
  try (ResultSet resultSet
           = statement.getResultSet()) { // SELECT statement: retrieve ResultS
    System.out.println("SELECT statement: processing ResultSet");
  }
} else {                                               // Update statement:
  int rowCount = statement.getUpdateCount();           // Retrieve the number
```

```
                                                       // rows affected.
  System.out.println("Update statement: Rows affected " + rowCount);
}
```

The method `executeQuery()` is used exclusively to execute a `SELECT` query. This method returns a `ResultSet` object that represents the rows retrieved as a result of executing the query. As a `ResultSet` is `AutoCloseable`, the method is called in a nested `try`-with-resources statement at (1) to ensure closure of the `ResultSet`.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {         // Obtain a Statement objec
  String sql = "select * from compositions";       // SQL query: select all ro

  try (ResultSet resultSet
           = statement.executeQuery(sql)) {    // (1) Nested try-with-resource
   System.out.println("Processing ResultSet");
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

The method `executeUpdate()` can be used to execute any SQL statement except a `SELECT` statement. An `int` value indicating the number of rows affected by the statement is returned as a result. The code below doubles the duration of each composition in the `compositions` table.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);     // (1)
     Statement statement = connection.createStatement()) {             // (2)
  String sql = "update compositions set duration = duration * 2";      // (3)
  int count = statement.executeUpdate(sql);                            // (4)
  System.out.println("Rows modified: " + count);
} catch (SQLException e) {
  e.printStackTrace();
}
```

The code above illustrates the procedure to use a basic statement. The numbered comments below correspond to the numbered lines in the code:

1. Create a `Connection` with the `DriverManager.getConnection()` method.
2. Obtain a `Statement` from the connection by calling its `createStatement()` method.
3. Formulate a SQL operation.
4. Call the appropriate execute method of the statement, passing

the SQL operation as the `String` parameter.

SQL Injection

Passing the SQL operation as a string in the execute method presents the risk of a *SQL injection*, as malicious SQL code can be injected into the query string. Assume that there is a basic statement that queries a table, where the query is dependent on a value:

Click here to view code image

```
ResultSet resultSet = statement.executeQuery(
    "select * from compositions where title like '" + value + "'");
```

A user may try to exploit this code by submitting a malicious value, such as the following:

Click here to view code image

```
String value = "SomeValue'; drop table compositions;";
```

The above query would be equivalent to the following statement, where the `SELECT` operation would be executed, followed by dropping the `compositions` table:

Click here to view code image

```
ResultSet resultSet2 = statement.executeQuery(
       "select * from compositions where title like '" +
       "SomeValue'; drop table compositions;");
```

To address this issue, one can use either a `PreparedStatement` object or the `Statement.enquoteLiteral()` method to sanitize the parameter value.

Prepared Statement

In contrast to the `Statement` object, which defines the SQL operation as a `String` parameter to the execute method, a `PreparedStatement` defines a SQL operation immediately at the point when the prepared statement is created with the `prepareStatement()` method. In other words, the SQL operation string is passed as an argument in the `prepareStatement()` method call, as shown at (1). The prepared statement executes the precompiled SQL operation at (2) by calling the `execute()` method.

Click here to view code image

```
String sql = "select * from compositions where duration > 200"; // SQL operati
try (Connection connection
         = DriverManager.getConnection(jdbcURL);                // Get connect
     PreparedStatement pStatement
         = connection.prepareStatement(sql)) {           // (1) Prepare statem
  boolean result = pStatement.execute();                 // (2) Execute
  System.out.println(result);
} catch (SQLException e) {
  e.printStackTrace();
}
```

Substitutional Parameterization

Consider the following code:

Click here to view code image

```
String sql = "select * from compositions where duration > ?"; // (1) SQL opera
                                                        // with 1 marker param
PreparedStatement pStatement
    = connection.prepareStatement(sql);                // (2) A prepared state
```

The SQL operation at (1) now contains a *marker parameter* (also known as *bind parameter* or *bind variable*). Prepared statements use *positional notation* where each question mark (`?`) indicates the position of a marker parameter starting with 1 from left to right. This could be any SQL operation, such as an `INSERT`, `UPDATE`, or `DELETE` statement with appropriate marker parameters. The SQL operation with marker parameters is passed as an argument to the `prepareStatement()` method to precom-pile this SQL operation into a prepared statement, as shown at (2).

However, before a prepared statement can be executed, all its parameters must be set—that is, all marker parameters must be substituted with a value. One can use the set methods of the `PreparedStatement` interface that are provided for all standard JDBC types to set the value of a marker parameter (see below).

Click here to view code image

```
pStatement.setInt(1,200);         // Sets the value of the marker parameter
                                  // at position 1 to 200.
```

Alternatively, the method `setObject()` can be used to supply a value and explicitly indicate the desired SQL type:

Click here to view code image

```
pStatement.setObject(1, 200, Types.INTEGER);
```

There is no requirement that parameters should be set in any specific order, as long as all parameters are set before executing the statement. Note that the position counter in JDBC is not 0-based as in the rest of the Java language, but starts at 1, which is typical in the database world. The driver converts the value of the Java data type in the method call to the corresponding value of the SQL data type. All parameter-setting methods throw a `SQLException` if the parameter index does not correspond to a marker parameter, or if it is called on a prepared statement that is already closed.

The `PreparedStatement` interface is a subinterface of the `Statement` interface. Once a prepared statement has been obtained with its precompiled SQL operation and values for its marker parameters have been supplied, the execute methods of the `PreparedStatement` interface can be used to execute the SQL operation. These execute methods are analogous to the corresponding methods in the `Statement` interface. Note that since the SQL operation to execute is already precompiled with the prepared statement, the execute methods of the `PreparedStatement` interface do not take any parameter. Specifying an argument will throw a `SQLException`, as will any marker parameter that is not substituted with a value.

```
boolean execute()
ResultSet executeQuery()
int executeUpdate()
```

The `PreparedStatement` interface provides the following set methods for setting the values of marker parameters:

Click here to view code image

```
void setString(int index, String value)
```

Sets a `String` value for the parameter designated by the index.

Click here to view code image

```
void setBoolean(int index, boolean value)
```

Sets a `boolean` value for the parameter designated by the index.

Click here to view code image

```
void setInt(int index, int value)
```

Sets an `int` value for the parameter designated by the index.

Click here to view code image

```
void setLong(int index, long value)
```

Sets a `long` value for the parameter designated by the index.

Click here to view code image

```
void setDouble(int index, double value)
```

Sets a `double` value for the parameter designated by the index.

Click here to view code image

```
void setBigDecimal(int index, BigDecimal value)
```

Sets a `BigDecimal` value for the parameter designated by the index.

Click here to view code image

```
void setDate(int index, Date value)
```

Sets a `Date` value for the parameter designated by the index.

Click here to view code image

```
void setObject(int index, Object value, int sqlType)
```

Sets any `Object` value for the parameter designated by the index, and the required SQL type which is a named constant in the `java.sql.Types` class—for example, `Types.INTEGER`, `Types.DATE`, or `Types.VARCHAR`.

Example 24.2 *Executing a Prepared Statement*

Click here to view code image

```
package dbDemo;
import java.sql.*;

public class XQTPreparedStatement {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    String sql = "select * from compositions where duration > ?";            //
    try (Connection connection = DriverManager.getConnection(jdbcURL);       //
        PreparedStatement pStatement = connection.prepareStatement(sql);) {  //
      pStatement.setInt(1, 200);                                             //
      boolean result = pStatement.execute();                                 //
      System.out.println(result);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output:

```
true
```

The procedure to use a prepared statement is illustrated by Example 24.2. The numbered comments below correspond to the numbered lines of code in the example.

1. Formulate a SQL operation with marker parameters.
2. Create a `Connection` with the `DriverManager.getConnection()` method.
3. Create a `PreparedStatement` from the connection by calling its `prepareStatement()` method and passing the SQL operation as a parameter.
4. Set the values of all marker parameters.
5. Call the appropriate execute method of the prepared statement.

Note that a prepared statement can be reused by substituting different values for its marker parameters. In the `try`-with-resources statement in Example 24.2, we can include the following code to change the condition in the `WHERE` clause at (1) by substituting another value for the marker parameter before executing the prepared statement:

Click here to view code image

```
pStatement.setInt(1, 100);
result = pStatement.execute();
```

Substitutional parameterization allows SQL operations to be precompiled and reused, leading to more efficient execution. Supplying values using marker parameters is more flexible, faster, and safer than using a basic statement because the prepared statement is precompiled and marker parameters are not prone to SQL injection.

Example 24.3 illustrates prepared statements to execute `INSERT`, `UPDATE`, and `DELETE` statements with marker parameters that are defined first. A database connection and three prepared statements are created in the header of the `try`-with-resources statement, respectively. The marker parameters are set in each prepared statement, before the statement is executed using the `executeUpdate()` method of the `Prepared-Statement` interface. The program prints three integers that indicate the number of rows affected by each SQL operation, which of course is dependent on the data in the `compositions` table.

Example 24.3 *Prepared Statement to Execute* `INSERT`*,* `UPDATE`*, and* `DELETE`

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;

public class PreparedStatementExecuteUpdate {
  public static void main(String[] args) {

    final String insSql = "insert into compositions VALUES(?, ?, ?)";
    final String updSql = "update compositions set title = ? where title = ?";
    final String delSql = "delete from compositions where duration = ?";

    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var pStatement1 = connection.prepareStatement(insSql);
        var pStatement2 = connection.prepareStatement(updSql);
        var pStatement3 = connection.prepareStatement(delSql)) {

        pStatement1.setInt(3, 150);
        pStatement1.setString(2, "Java Jazz");
        pStatement1.setString(1, "ushm91736991");
        int result1 = pStatement1.executeUpdate();
        System.out.println(result1);

        pStatement2.setString(1, "Java Jive");
        pStatement2.setString(2, "Java Jazz");
        int result2 = pStatement2.executeUpdate();
        System.out.println(result2);

        pStatement3.setInt(1, 200);
        int result3 = pStatement3.executeUpdate();
        System.out.println(result3);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output:

```
1
1
0
```

Callable Statement

The `Callable` interface is a subinterface of the `PreparedStatement` interface. Callable statements are very similar to prepared statements because both use marker parameters. The main difference is that callable statements are used to invoke *named* *stored procedures and stored functions that reside on the database side*. The difference between a function and a procedure is that a function is intended to return a value. Both are precompiled code on the database side and can be invoked via a callable statement on the application side.

Syntax of Calling Stored Procedures and Functions

The syntax for calling stored procedures and functions is shown below at (1) and (2), respectively. The parameter list of a stored procedure or function can have any number of marker parameters, but the value returned by a function call is assigned to the first marker on the left-hand side of the `=` sign.

Click here to view code image

```
String procedureCall
    = "{ call some_procedure(?,?) }";           // (1) Call a stored procedure.
String functionCall = "? = { call some_function(?) }"; // (2) Call a function.
CallableStatement cStatement1 = connection.prepareCall(procedureCall); // (3)
CallableStatement cStatement2 = connection.prepareCall(functionCall);  // (4)
```

Callable statements are prepared analogous to prepared statements by calling the `prepareCall()` factory method of the `Connection` interface, as shown at (3) and (4). The marker parameters in the call must be set up in accordance with the parameters of the stored procedure or function before executing the callable statement.

Three kinds of marker parameters can be specified in a call to a stored procedure or function on the database side: IN, OUT, and INOUT parameters.

IN Parameters

These parameters pass values to the stored procedure or function. These are marker parameters whose values are initialized with the inherited `set`*`XXX`*`()` methods from the superinterface `PreparedStatement`. The IN parameters for callable statements are handled exactly as those for prepared statements—a prepared statement has only IN parameters. The parameter index starts at 1. It goes without saying that IN parameters must be initialized before the callable statement is executed.

Click here to view code image

```
cStatement1.setString(1, "Hi");              // Set value of IN parameter
                                             // at marker 1
cStatement2.setObject(2, 42, Types.INTEGER); // Set value of IN parameter
                                             // at marker 2
```

OUT Parameters

These parameters are marker parameters that hold values returned by a stored procedure or a function. Before executing callable statements with OUT parameters, the `registerOutputParameter()` method of the callable statement should be used to specify the expected types of the values returned in the OUT parameters.

Click here to view code image

```
cStatement1.registerOutParameter(2, Types.VARCHAR); // Register SQL VARCHAR as
                                                    // type for the
                                                    // 2nd marker parameter.
cStatement2.registerOutParameter(1, Types.INTEGER); // Register SQL INTEGER as
                                                    // type for the

                                                    // 1st marker parameter.
```

> Click here to view code image
>
> ```
> void registerOutParameter(int parameterIndex, int sqlType)
> ```
>
> Registers the type of the OUT parameter at the marker parameter index to be of the specified JDBC type. The JDBC type is a constant specified by `java.sql.Types`—for example, `Types.INTEGER` and `Types.VARCHAR`.

Note there can be several OUT parameters, and the *first* marker parameter of a stored function call must also register the expected type of its return value.

The `Callable` interface inherits the execute methods of the `PreparedStatement` interface. The code below executes callable statements.

Click here to view code image

```
boolean result1 = cStatement1.execute();
boolean result2 = cStatement2.execute();
```

After execution, the `get`*`XXX`*`()` methods of the `CallableStatement` interface can be used to retrieve OUT parameter values of relevant types, based on the parameter index.

Click here to view code image

```
String result1 = cStatement1.getString(2);      // Retrieve value of OUT parame
                                                // at marker 2.
```

The generic method `getObject()` can also be used to retrieve OUT parameter values, specifying both an index and the expected Java type of the value (see below).

Click here to view code image

```
String result2 = cStatement2.getObject(1, Integer.class);// Retrieve value of
                                                         // parameter at marke
```

Following are selected get methods in the `CallableStatement` interface to retrieve values of OUT parameters:

> Click here to view code image
>
> ```
> String getString(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as a `String`.
>
> Click here to view code image
>
> ```
> int getBoolean(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as a `boolean`.
>
> Click here to view code image
>
> ```
> int getInt(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as an `int`.
>
> Click here to view code image
>
> ```
> double getLong(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as a `long`.
>
> Click here to view code image
>
> ```
> double getDouble(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as a `double`.
>
> Click here to view code image
>
> ```
> BigDecimal getBigDecimal(int paramIndex)
> ```
>
> Retrieves the value of the designated marker parameter as a

`BigDecimal.`

```
Date getDate(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `Date.`

Click here to view code image

```
<T> T getObject(int paramIndex, Class<T> type)
```

Retrieves the value of the designated marker parameter as an object of the indicated Java type.

INOUT Parameters

An INOUT parameter acts as both an IN parameter to pass a value to the stored procedure or function and an OUT parameter to return a value from the stored procedure or function. Before the execution of the callable statement, its value can be initialized with a `set`*`XXX`*`()` method as for an IN parameter and can register the JDBC type of the return value by calling the `registerOutputParameter()` method as for an OUT parameter. The returned values can be retrieved using the appropriate `get`*`XXX`*`()` method as we have seen for OUT parameters.

Example 24.4 illustrates the basics of calling stored procedures and functions. The nitty-gritty of creating and deploying stored methods and procedures is database specific, and beyond the scope of this book.

Example 24.4 *Stored Procedures and Functions*

Click here to view code image

```
public void storedProcedureCall(Connection connection) {
  final String callProc
      = "{call longCompositionsProc(?, ?)}";             // (1) 1: IN 2:OUT
  try (CallableStatement cStmt = connection.prepareCall(callProc)) { // (2)
    int duration = 100;                                              // (3)
    cStmt.setInt(1, duration);                                       // (4)
    cStmt.registerOutParameter(2, Types.INTEGER);                    // (5)
    cStmt.execute();                                                 // (6)
    int returnedValue = cStmt.getInt(2);                             // (7)
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {
    e.printStackTrace();
  }
}

public void storedFunctionCall(Connection connection) {
  final String callFunc
      = "? = {call longCompositionsFunc(?)}";             // (8) 1:OUT 2:IN
  try (CallableStatement cStmt = connection.prepareCall(callFunc)) {
    cStmt.registerOutParameter(1, Types.INTEGER);                     // (9)
    int duration = 100;
    cStmt.setInt(2, duration);                                        // (10
    cStmt.execute();
    int returnedValue = cStmt.getInt(1);                              // (11
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {

    e.printStackTrace();
  }
}
```

The two methods `storedProcedureCall()` and `storedFunctionCall()`, in Example 24.4, call a stored procedure and a stored function, respectively. Both the stored procedure named `longCompositionsProc` and the stored function named `longCompositionsFunc` compute the number of rows in the `compositions` table whose duration is greater than the duration specified in the call. We assume that the stored procedure and function are implemented and deployed on the database side. The result returned will depend on the state of the `compositions` table. The numbered comments below correspond to the numbered lines in Example 24.4.

1. The stored procedure `longCompositionsProc` has one IN and one OUT parameter designated by the marker positions 1 and 2, respectively.
2. The `prepareCall()` method prepares a callable statement for the stored procedure call.
3. The variable `duration` is declared and initialized with the value `100`.
4. The IN parameter at marker 1 is initialized with the value of the variable `duration` (`100`). We are interested in finding the number of rows with duration greater than 100.
5. The OUT parameter at marker 2 is registered to return a value of JDBC type `INTEGER` which will be retrieved as a value of Java type `int`.
6. The `execute()` method executes the stored procedure call in the callable statement.
7. After execution, the `int` value returned in the OUT parameter at marker 2 can be retrieved. Of course, this value depends on the state of the database.
8. The stored function `longCompositionsCall` has one IN and one OUT parameter designated by the marker positions 2 and

1, respectively.

9. Note that the OUT parameter at marker 1 is reserved for the return value from the stored function call, but its JDBC type must be registered, in this case JDBC type `INTEGER`.
10. The IN parameter at marker 2 is initialized with the value `100` as the lower limit for duration.
11. After execution, the returned value at marker 1 can be retrieved.

24.5 Processing Query Results

A *result set* represents a table of rows that is the result of executing a database query. By default, it is not updatable—that is, it cannot be modified. `SELECT` queries can be executed by the different types of statements that we have encountered so far to create result sets (p. 1522). It is quite obvious that basic and prepared statements can produce such results, but it is also possible to get a result set as an output value from a stored procedure or function.

The first thing to note is that a `ResultSet` object is `AutoCloseable`. Of course, it will be automatically closed and its resources released if handled in a `try`-with-resources statement. However, it can be prudent to call its `close()` method to release its resources immediately after use rather than wait for it to happen automatically. This might improve overall performance and facilitate scalability of the application.

Traversing the Result Set

Associated with a result set is a *cursor* that indicates the current row in the table of rows associated with the result set. This cursor should not be confused with a *database cursor* which represents a `ResultSet` object itself in database terminology. Initially the cursor is positioned just before the first row. A result set can be traversed by moving the cursor forward by one row by calling the `next()` method of the `ResultSet` interface.

- If the `next()` method returns `true`, the cursor has been moved forward and the row it points to is now the current row whose column values can be read from the result set using the get methods of the `ResultSet` interface (p. 1537).
- If the `next()` method returns `false`, the cursor now points

past the last row and there are no more rows to traverse in the result set.

A result set can only be traversed once from the first row to the last row. The `next()` method can be used as the condition of a `while` loop to process all rows in the result set.

```
while (resultSet.next()) {
  /* Process current row. */
}
```

An `if` statement can be used to determine whether the result set is empty or not. For example, a `SELECT` statement parameterized with a primary key value would expect to return a single row. However, it is also possible that no rows are returned if the marker parameter specifies a nonexistent key value. In any case, the `next()` method should be invoked to check if there are any rows in the result set.

```
if (resultSet.next()) {
  /* Result set not empty. */
}
```

Result Set Navigation Methods

The following methods of the `ResultSet` interface can be used to navigate a result set. By default, the cursor can only be moved in the forward direction by calling the `next()` method. The other navigation methods (`previous()`, `next()`, `first()`, `last()`, `absolute()`, and `relative(int row)`) can only be used if scrolling is first enabled by setting the appropriate `ResultSet type` (Table 24.3, p. 1539).

> ```
> boolean next()
> ```
>
> Returns `true` if it is possible to move the cursor forward by one row from its current position; otherwise, it returns `false`. The return value `false` implies that either the cursor is after the last row or the result set is empty. Calling the `next()` method after it returns `false` will throw a `SQLException`.
>
> ```
> boolean previous()
> ```
>
> Returns `true` if it is possible to move the cursor to the previous row from its current position; otherwise, it returns `false`. The return value `false` implies that either the cursor is before the first row or the result set is empty. In both case, the cursor does not point to a valid row—in which case, calling a

method on the `ResultSet` that requires a valid row will throw a `SQLException`.

```
boolean first()
```

Returns `true` if it is possible to move the cursor to the first row; otherwise, it returns `false`. The return value `false` implies that the result set is empty.

```
boolean last()
```

Returns `true` if it is possible to move the cursor to the last row; otherwise, it returns `false`. The return value `false` implies that the result set is empty.

Click here to view code image

```
boolean absolute(int rowNumber)
```

If the specified row number is positive, the cursor is moved to the absolute row number with respect to the beginning of the result set. Calling `absolute(1)` is the same as calling `first()`.
If the given row number is negative, the cursor is moved to the absolute row number with respect to the end of the result set. Calling `absolute(-1)` is the same as calling `last()`.
If the row number is zero, the cursor is moved to before the first row.
An attempt to position the cursor beyond the first or the last row in the result set leaves the cursor before the first row or after the last row, respectively.

```
boolean relative(int rows)
```

A positive/negative value in rows moves the cursor that many rows forward/ backward from the current position, respectively. Attempting to move beyond the first or the last row in the result set will set the cursor before the first row or after the last row, respectively.
Calling the method `relative(1)` is the same as calling the method `next()`.
Calling the method `relative(-1)` is the same as calling the method `previous()`.
Calling `relative(0)` is valid, but does not change the cursor position.

Optimizing the Fetch Size

To optimize the processing of rows in the result set that resulted from executing a query, only a certain number of these rows are fetched at a time from the database when they are needed by the result set. The default number to fetch, called the default *fetch size*, is set by the `Statement` object that created the result set.

The fetch size achieves a balance between the time it takes to fetch a row (depending on the row size), the total number of rows, and the number of round trips necessary between the database and the application to fetch all the rows.

However, we can improve the performance of processing the result set by providing a hint about the fetch size in the `setFetchSize()` method. Setting the fetch size only provides the JDBC driver with a hint as to the number of rows that should be fetched at a time from the database when more rows are needed by the result set. Setting the fetch size value to `0` is ignored by the JDBC driver, and it will then decide what fetch size to use. The fetch size can be set at the level of the `Statement` or the `ResultSet`.

Click here to view code image

```
PreparedStatement statement
    = connection.prepareStatement("select * from compositions");
statement.setFetchSize(20);            // Rows would be downloaded 20 at a tim
ResultSet resultSet = statement.executeQuery();
while (resultSet.next()){
 /* Rows are processed one at a time regardless of the fetch size */
  if (/* some condition */) {
    resultSet.setFetchSize(10);        // Rows would be downloaded 10 at a tim
  }
}
```

Calling the `next()` method will still only move the cursor to the next row, if there are still rows in the result set to traverse, regardless of the actual fetch size that is set. In other words, the program logic does not change in any way regardless of the fetch size optimization.

Extracting Column Values from the Current Row

The column values of the current row indicated by the cursor can be extracted using the get methods provided by the result set (see below).

Click here to view code image

```
String strValue1 = resultSet.getString(1);              // Using column inde
String strValue2 = resultSet.getString("column_name");    // Using column name
```

We can also use the `getObject()` generic method, specifying either the column index or the column name together with the corresponding SQL type.

Click here to view code image

```
String strValue3 = resultSet.getObject(1, String.class);
String strValue4 = resultSet.getObject("column_name", String.class);
```

If the result set is not processed in a `try`-with-resources statement, consider closing the result set explicitly as soon as possible, after it has been processed.

```
resultSet.close();
```

Result Set Methods to Extract Column Values of the Current Row

The following get methods of the `ResultSet` interface can be used to extract the column values of the current row indicated by the cursor in the result set. The column index or the column label can be passed as a parameter to designate the appropriate column in the table. The `CallableStatement` interface provides analogous `get`*`XXX`*`()` methods to retrieve values of OUT parameters (p. 1530).

Selected get methods from the `ResultSet` interface to extract column values of the current row are shown below:

Click here to view code image

```
String getString(int columnIndex)
String getString(String columnLabel)
```

Retrieve the value of the column as a `String`.

Click here to view code image

```
int getBoolean(int columnIndex)
int getBoolean(String columnLabel)
```

Retrieve the value of the column as a `boolean`.

Click here to view code image

```
int getInt(int columnIndex)
int getInt(String columnLabel)
```

Retrieve the value of the column as an `int`.

Click here to view code image

```
double getLong(int columnIndex)
double getLong(String columnLabel)
```

Retrieve the value of the column as a `long`.

Click here to view code image

```
double getDouble(int columnIndex)
double getDouble(String columnLabel)
```

Retrieve the value of the column as a `double`.

Click here to view code image

```
BigDecimal getBigDecimal(int columnIndex)
BigDecimal getBigDecimal(String columnLabel)
```

Retrieve the value of the column as a `BigDecimal`.

Click here to view code image

```
Date getDate(int columnIndex)
Date getDate(int columnLabel)
```

Retrieve the value of the column as a `Date`.

Click here to view code image

```
<T> T getObject(int columnIndex, Class<T> type)
<T> T getObject(int columnLabel, Class<T> type)
```

Retrieve the value of the column as an object of the indicated Java type.

Example 24.5 *Processing a* `ResultSet`

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.time.Duration;

import static java.lang.System.out;

public class ResultSetProcessing {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    final String sql = "select * from compositions where duration > ?";    //
    try (var connection = DriverManager.getConnection(jdbcURL);            //
        var pStatement = connection.prepareStatement(sql);) {              //
      pStatement.setInt(1, 0);                                             //
      var resultSet = pStatement.executeQuery();
      try (resultSet) {                                                    //
```

```
        while (resultSet.next()) {                                         //
          String isrc = resultSet.getString(1);                            //
          String title = resultSet.getObject(2, String.class);             //
          int duration = resultSet.getInt("duration");                     //
          out.println("[" + isrc + ", " + title + ", " + duration + "]");  //
        }
      } // Closes the result set.
    } catch (SQLException e) {                                             //
      e.printStackTrace();
    } // Closes the prepared statement and the connection.
  }
}
```

Probable output from the program:

Click here to view code image

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
[ushm91736699, Why Don't, 178]
```

Example 24.5 illustrates the main steps in the interaction between a Java application and a relational database. The example uses the `musicDB` database that has been used so far to illustrate salient features of programming this interaction. The numbered comments below correspond to the numbered code lines in Example 24.5.

1. Define the `SELECT` statement to execute, having one marker parameter for the second operand of the `boolean` expression.
2. Use the `try`-with-resources statement to declare the resources and to close the connection and the prepared statement afterward. Obtain the connection to the database.
3. Create the prepared statement in the header of the `try`-with-resources statement.
4. Substitute the marker parameter with a value in the `SELECT` statement at (1).
5. Use the nested `try`-with-resources statement to close the result set after processing it. The prepared statement is executed in the header of the `try`-with-resources statement, returning a result set. Any exception thrown will be propagated to the `catch` clause at (11) of the outer `try`-with-resources statement.
6. The `while` loop processes the rows in the result set.
7. Extract the column values of the current row denoted by the result set cursor by calling the appropriate get methods of the result set. Extract a string (`isrc`) from column 1 using the `getString()` method.
8. Extract a string (`title`) from column 2 using the `getObject()` method.

9. Extract an `int` value (`duration`) by column name using the `getInt()` method.
10. Print the column values of the current row.
11. Use the `catch` clause of the outer `try`-with-resources statement to catch any `SQLException`.

24.6 Customizing Result Sets

It is possible to customize certain features of the result set. Such features may not necessarily be available across all databases and in some cases may result in performance degradation. In this section we discuss possible customizations and what can be achieved by them.

The following features of the result set can be customized:

- *Result set type*: This feature allows customization of the *navigational direction* of the result set traversal and the *sensitivity* of the result set to reflect changes made in the underlying data while it remains open.
- *Result set concurrency*: This feature allows customization of the *updatability* of the result set—that is, whether the `ResultSet` object can be updated using the `ResultSet` interface.
- *Result set holdability*: This feature allows customization of whether the result set is retained or closed when the current transaction is committed.

Valid values for these features are defined by static constants in the `ResultSet` interface, shown in Table 24.3.

Table 24.3 *Selected Constants Defined in the* `ResultSet` *Interface*

Result set type constants	**Description**
`TYPE_FORWARD_ONLY`	The type for a `ResultSet` object whose cursor may only move *forward*. This is the default `ResultSet` type.
`TYPE_SCROLL_INSENSITIVE`	The type for a `ResultSet` object that is *scrollable* (i.e., the cursor can move forward and backward), but generally *not sensitive* to changes made to the underlying data while the `ResultSet` is open.
`TYPE_SCROLL_SENSITIVE`	The type for a `ResultSet` object that is *scrollable* and generally *sensitive* to changes made to the underlying data while the `ResultSet` is open.
Result set concurrency consta	**Description**

`CONCUR_READ_ONLY`	The concurrency for a `ResultSet` object that may *not* be updated—that is, it is read-only. In other words, *updatability* of the `ResultSet` object is not allowed. This is the default `ResultSet` concurrency.
`CONCUR_UPDATABLE`	The concurrency for a `ResultSet` object that may be updated. That is, *updatability* of the `ResultSet` is allowed.
Result set holdability constan	**Description**
`CLOSE_CURSORS_AT_COMMIT`	Open `ResultSet` objects with this *holdability* will be *closed* when the current transaction is committed. Such a `ResultSet` is said *not to be holdable*.
`HOLD_CURSORS_OVER_COMMIT`	Open `ResultSet` objects with this *holdability* will remain *open* when the current transaction is committed. Such a `ResultSet` said to be *holdable*.

Result Set Type

The *result set type* refers to the direction of navigation in the result set and whether changes in the underlying data get reflected in an open result set.

By default, the result set type is set to `ResultSet.TYPE_FORWARD_ONLY`, which makes the cursor move in the forward direction only, from before the first row and successively to after the last row. This essentially means that the traversal of the rows in the result set is only possible with the `next()` method. This option is considered to be a safe default choice for all result sets because forward-only progression through rows does *not* require the Java application to download the whole *database cursor* (i.e., all rows that comprise the result of the query on the database). It also allows the fetch size to be customized to achieve better performance, especially in a situation where the query might result in a large number of rows.

With the option `ResultSet.TYPE_FORWARD_ONLY`, the result set may reflect changes in the underlying database that occur while the result set is being processed. Not all databases are capable of reflecting changes.

Two other options (`ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_ SENSITIVE`) are also available for the result set type, both of which enable scrolling— that is, the arbitrary navigation direction in the result set using the `previous()`, `next()`, `first()`, `last()`, `absolute()`, and `relative()` methods (p. 1534). The cursor can move both forward and backward relative to the current position, or to an absolute position. In order to jump to a particular row in the result set, the entire result set may need to be downloaded first from the database, which can result in performance degradation when dealing with very large result sets returned by the query.

The options `ResultSet.TYPE_SCROLL_INSENSITIVE` and `ResultSet.TYPE_SCROLL_SENSITIVE` make the result set *insensitive* and *sensitive*, respectively, to changes made in the underlying data source while the result set is open.

The `DatabaseMetaData.supportsResultSetType()` method can be called to determine whether a particular result set type is actually supported by the database (p. 1543).

Result Set Concurrency

The *result set concurrency* feature enables or disables whether the result set can be updated or not—that is, it indicates the *concurrency mode* of the result set. By default, the result set concurrency is set to `ResultSet.CONCUR_READ_ONLY`, which means the result set cannot be updated. Alternatively, updatability can be enabled by setting the result set concurrency to be `ResultSet.CONCUR_UPDATABLE`. This allows the result set to be modified.

The `DatabaseMetaData.supportsResultSetConcurrency()` method can be called to identify valid combinations of result set type and result set concurrency supported by the current JDBC driver (p. 1543).

Result Set Holdability

The *result set holdability* refers to whether the result set may remain open or be closed when the current transaction is committed (p. 1545). The default value for the result set holdability is actually database dependent. One option is `ResultSet.HOLD_CURSORS_OVER_COMMIT`, which allows the result sets (cursors) to remain open when the transaction is committed.

Alternatively, holdability can be set to `ResultSet.CLOSE_CURSORS_AT_COMMIT`, which indicates that `ResultSet` objects (cursors) should be closed upon commit. Closing cursors can result in better performance for some applications.

The `DatabaseMetaData.supportsResultSetHoldability()` method can be called to identify the holdability supported by the current JDBC driver (p. 1543).

Please note that not all databases and JDBC drivers support these customizations. Such customizations can be specified when the statement object is created by calling the relevant factory method of the `Connection` interface. Analogous methods with the same result set customizations are also available for a `Statement` and a `Callable`.

The following overloaded method is defined in the `Connection` interface:

Click here to view code image

```
PreparedStatement prepareStatement(String sql,
                          int resultSetType, int resultSetConcurrency,
                          int resultSetHoldability)
```

Example 24.6 illustrates using different `ResultSet` options. Note that in this example, the call to the `updateRow()` method will result in the column value of the underlying row in the database to be updated with the column value of the current row in the result set which was just updated. In other words, the `updateRow()` method commits the changes to the database and thereby the result set is also closed as we have specified `ResultSet.CLOSE_CURSORS_AT_COMMIT`. However, if the automatic commit is disabled, as shown at (1), the changes in the database due to the `updateRow()` method will not be committed and the result set will not be closed until an explicit call to the `commit()` method is executed, as shown at (2) (p. 1545).

Example 24.6 *Using* `ResultSet` *Options*

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;

import java.sql.ResultSet;
import java.sql.SQLException;
public class ResultSetCustomization {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var statement = connection.prepareStatement(
            "select duration from compositions where title = ?",
            ResultSet.TYPE_FORWARD_ONLY,        // Forward direction. May refl
                                                // database changes.
            ResultSet.CONCUR_UPDATABLE,         // Result set is updatable.
            ResultSet.CLOSE_CURSORS_AT_COMMIT   // Result set is closed on com
            )) {
      connection.setAutoCommit(false);          // (1) Disables automatic comm
      statement.setString(1,"Vacation");
```

```
        try (ResultSet resultSet = statement.executeQuery();) {
          if (resultSet.next()) {                     // Moves forward one row.
            resultSet.updateInt("duration", 147); // Updates the current row
                                                  // in the result set.
            resultSet.updateRow();                // Updates the underlying
                                                  // database.
            System.out.println("Updated");
          }
          connection.commit();                    // (2) Also closes the result
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }
```

Possible program output:

```
Updated
```

24.7 Discovering Database and `ResultSet` Metadata

The JDBC API allows Java programs to interact with many different types of databases. Obviously, different database providers have different capabilities and default behaviors, and may or may not support certain SQL features. It is possible to obtain such information using the `java.sql.DatabaseMetaData` object. This object is obtained from the JDBC `Connection` and can be used to investigate various capabilities and many other properties, such as support for type, concurrency, and holdability of a result set.

Following selected methods from the `DatabaseMetaData` API can be used to examine the capabilities of a database:

> Click here to view code image
>
> ```
> String getDatabaseProductName()
> ```
>
> Retrieves the name of this database product.
>
> Click here to view code image
>
> ```
> String getDatabaseProductVersion()
> ```
>
> Retrieves the version number of this database product.
>
> ```
> String getSQLKeywords()
> ```
>
> Retrieves a comma-separated list of all SQL keywords

> supported by this database.
>
> Click here to view code image
>
> ```
> boolean supportsResultSetType(int resultSettype)
> ```
>
> Retrieves whether this database supports the given result set type.
>
> Click here to view code image
>
> ```
> boolean supportsResultSetHoldability(int holdability)
> ```
>
> Retrieves whether this database supports the given result set holdability.
>
> Click here to view code image
>
> ```
> boolean supportsResultSetConcurrency(int resultSettype, int concurrency)
> ```
>
> Retrieves whether this database supports the given combination result set type/concurrency type.

In addition to the database metadata, a `ResultSetMetaData` object can be used to discover information about the structure of a given result set.

Following selected methods from the `ResultSetMetaData` API can be used to examine the structure of the result set:

> ```
> int getColumnCount()
> ```
>
> Returns the number of columns in this result set.
>
> Click here to view code image
>
> ```
> String getColumnName(int column)
> ```
>
> Returns the name of the designated columns in this result set.
>
> ```
> int getColumnType(int column)
> ```
>
> Returns the type of the designated columns in this result set. The `int` value returned corresponds to the SQL type designated by constants in `java.sql.Types`. The values `12` and `4` represent the SQL types `VARCHAR` and `INTEGER`, respectively.

Example 24.7 prints various metadata about the database and the result set.

Example 24.7 *Discovering Metadata for the Database and* `ResultSet`

Click here to view code image

```
package dbDemo;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DBMetadata {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // Print various information about the database:
      DatabaseMetaData dbMetaData
          = connection.getMetaData();                 // Obtain DatabaseMetaDa
      String dbName               = dbMetaData.getDatabaseProductName();
      String dbVersion            = dbMetaData.getDatabaseProductVersion();
      String sqlKeywords          = dbMetaData.getSQLKeywords();
      boolean forwardOnly = dbMetaData.supportsResultSetType(
          ResultSet.TYPE_FORWARD_ONLY);
      boolean cursorOpen  = dbMetaData.supportsResultSetHoldability(
          ResultSet.HOLD_CURSORS_OVER_COMMIT);
      boolean forwardUpdate = dbMetaData.supportsResultSetConcurrency(
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_UPDATABLE);

      System.out.println("Various info about the database:");
      System.out.println("Database name: " + dbName);
      System.out.println("Version: " + dbVersion);
      System.out.println("SQL keywords: " + sqlKeywords);
      System.out.println("TYPE_FORWARD_ONLY: " + forwardOnly);
      System.out.println("HOLD_CURSORS_OVER_COMMIT: " + cursorOpen);
      System.out.println("TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: "
          + forwardUpdate);

      // Create a ResultSet and print its structure:
      String sql = "select * from compositions where duration > ?";
      try (var pStatement = connection.prepareStatement(sql);) {
        pStatement.setInt(1, 100);
        var resultSet = pStatement.executeQuery();
        try (resultSet){
          System.out.println("Structure of ResultSet:");
          ResultSetMetaData rsMetaData
              = resultSet.getMetaData();              // Obtain ResultSetMetad
          int columnCount = rsMetaData.getColumnCount();
          System.out.println("Number of columns:" + columnCount);
          for (int i = 1; i <= columnCount; i++){
            String name = rsMetaData.getColumnName(i);
            int type = rsMetaData.getColumnType(i);   // Value of Types consta
            System.out.println(name + ": " + type);
          }
        }
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
Various info about the database:
Database name: Apache Derby
Version: 10.15.2.0 - (1873585)
SQL keywords: ALIAS,BIGINT,BOOLEAN,CALL,CLASS,COPY,DB2J_DEBUG,EXECUTE,… (EDITE
TYPE_FORWARD_ONLY: true
HOLD_CURSORS_OVER_COMMIT: true
TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: true
Structure of ResultSet:
Number of columns:3
ISRC: 12
TITLE: 12
DURATION: 4
```

24.8 Implementing Transaction Control

The JDBC protocol allows the application to control *database transactions*. A *transaction* is a set of statements that is executed as a logical unit. Either all changes due to the execution of statements in the transaction are committed to the database, or none of the changes made in the transaction are committed. A *commit* results in the changes made in the transaction being made permanent in the database. A commit thereby ends a transaction. A *rollback* results in undoing all changes made in the transaction—that is, restoring the database state to what it was before the transaction commenced. A *savepoint* defines a logical rollback point *within* a transaction. A *rollback* can be used to undo all changes made since a savepoint was set up in the transaction—that is, restoring the database state to what it was before the save-point was set up.

By default, JDBC connections are in *auto-commit mode*, which means that commit occurs automatically when the statement processing completes successfully. To change this behavior, the following method call can be used:

Click here to view code image

```
connection.setAutoCommit(false);
```

Once auto-commit is turned off, transactions can be controlled using the `rollback()` and `commit()` methods of the `Connection` interface. Each transaction can include any number of insert, update, and delete actions and must end in either a commit that would make all pending transaction changes permanent, or a rollback if the pending transaction changes should be discarded. Some databases also support save-points, which provides an ability to partially roll back a transaction, discarding only those pending changes that occurred in the transaction after the savepoint was set up. Whether a given database supports savepoints can be determined by calling the `DatabaseMetaData.supportsSavepoints()` method. It is important to remember that normal closing of a connection implicitly causes the transaction to commit. However, the transaction will be rolled back if the program terminates with an uncaught exception.

If an exception is intercepted, the Java runtime considers this exception to be successfully handled and therefore resumes normal execution after the catch block. It is worth reciting the mantra that the JDBC interfaces, including the `Connection` interface, implement `AutoCloseable`, which means that all statements and connections are closed implicitly following the execution of the `try`-with-resources statement. This implies that if the transaction was not explicitly rolled back inside the exception handler, then by default, the transaction will be committed.

Following selected methods of the `java.sql.Connection` interface can be used to control transactions:

> Click here to view code image
>
> ```
> void setAutoCommit(boolean autoCommit)
> ```
>
> Enables or disables *auto-commit mode* of this connection. The value `true` enables auto-commit mode and `false` disables it.
>
> Click here to view code image
>
> ```
> Savepoint setSavepoint()
> Savepoint setSavepoint(String name)
> ```
>
> Create an unnamed or a named savepoint in the current transaction and returns the new `Savepoint` object that represents it, respectively.
>
> Click here to view code image
>
> ```
> void rollback()
> void rollback(Savepoint savepoint)
> ```
>
> Undo all changes made in the current transaction or all changes made since the named savepoint was set up in the current transaction, respectively.
>
> ```
> void commit()
> ```
>
> Makes all changes made since the previous commit/rollback

permanent.

Example 24.8 *Controlling Transactions*

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Savepoint;

public class Transactions {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // SQL statements:
      final String insSql = "insert into compositions VALUES(?, ?, ?)";
      final String updSql = "update compositions set title = ? where title = ?
      final String delSql = "delete from compositions where duration = ?";

      // Create statements:
      try (var insStatement = connection.prepareStatement(insSql);
          var updStatement = connection.prepareStatement(updSql);
          var delStatement = connection.prepareStatement(delSql);) {

        connection.setAutoCommit(false);            // (1) Auto-commit disable
        insStatement.setInt(3, 150);                // (2) Insert a new row.
        insStatement.setString(2, "Java Jazz");
        insStatement.setString(1, "ushm91736991");
        int insResult = insStatement.executeUpdate();
        System.out.println("INSERT: " + insResult);

        updStatement.setString(1, "Java Jive");     // (3) Update an existing
        updStatement.setString(2, "Rage");
        int updResult = updStatement.executeUpdate();
        System.out.println("UPDATE: " + updResult);

        Savepoint savePoint = connection.setSavepoint(); // (4) Set a savepoin
        delStatement.setInt(1, 178);                     // (5) Delete a row.
        int delResult = delStatement.executeUpdate();
        System.out.println("DELETE: " + delResult);

        connection.rollback(savePoint);   // (6) Roll back to safepoint.
        connection.commit();              // (7) Commits only (2) and (3).
      } catch (SQLException e) {
        connection.rollback();            // (8) Roll back any changes.
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Output from the program:

```
INSERT: 1
UPDATE: 1
DELETE: 1
```

Example 24.8 illustrates how transactions can be controlled with commit and rollback operations. The numbered comments below correspond to the numbered code lines in Example 24.8.

The following rows are in the `compositions` table at the start.

Click here to view code image

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
[ushm91736699, Why Don't, 178]
```

1. Auto-commit is disabled. A new transaction starts.
2. A new row is inserted:

   Click here to view code image

   ```
   [ushm91736991, Java Jazz, 150}
   ```

3. The title `"Rage"` of an existing row:

   ```
   [ushm91736698, Rage, 308]
   ```

   is updated to `"Java Jive"`:

   Click here to view code image

   ```
   [ushm91736698, Java Jive, 308]
   ```

4. A savepoint is set.
5. The row with the duration `178` is deleted:

   Click here to view code image

   ```
   [ushm91736699, Why Don't, 178]
   ```

6. A rollback is performed, rolling back the deletion of the row at (5) that was performed since the last savepoint was set up at (4).
7. A commit is performed, committing the insertion at (2) and the update at (3), leaving the database in the following state and ending the transaction that started at (1):

   Click here to view code image

   ```
   [ushm91736697, Vacation, 231]
   [ushm91736698, Java Jive, 308]
   [ushm91736699, Why Don't, 178]
   [ushm91736991, Java Jazz, 150]
   ```

8. In case a `SQLException` is thrown in the inner `try`-with-resources statement, all changes are rolled back.

Review Questions

24.1 Which of the following statements is true about the JDBC `ResultSet`?

Select the one correct answer.

- **a.** The method `next()` throws an exception when no rows were returned by the query.
- **b.** The method `next()` throws an exception when it is invoked more times than the number of rows in the result set.
- **c.** The method `absolute(1)` navigates to the first row of the result set, if there are rows in the result set.
- **d.** The method `absolute(0)` navigates to the first row of the result set, if there are rows in the result set.

24.2 What is the correct order in which the following JDBC resources are closed?

Select the one correct answer.

- **a.** `Connection, Statement, ResultSet`
- **b.** `Statement, ResultSet, Connection`
- **c.** `ResultSet, Statement, Connection`
- **d.** Closure order is irrelevant.
- **e.** Closure order is irrelevant when auto-commit mode is enabled.

24.3 Assume that a database has the following `questions` table with the specified columns and rows of data:

question		
(VARCHAR) NOT NULL		
`What was Deep Thought's answer?`		
`Where is Wally?`		
`Is Waldo younger than Wally?`		
`Which the one Wally? bobble hat.`		
`Right here?`		

What will be the result of executing the following code (assuming that `jdbcUrl`, `username`, and `password` are correctly initialized)?

Click here to view code image

```
String findAnswers = "select * from questions where question like ?";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
     PreparedStatement ps = c.prepareStatement(findAnswers)) {
  ps.setString(1, "Where%");
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      String question = rs.getObject(2, String.class);    // (1)
      String answer = rs.getObject(3, String.class);      // (2)
      answer = (answer == null) ? "No answer." : answer; // (3)
      System.out.println(question + " - " + answer);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

Select the one correct answer.

a. The program will print one line.
b. The program will print two lines.
c. The program will throw an exception at (1).
d. The program will throw an exception at (2).
e. The program will throw an exception at (3).

24.4 Assume that a database has the following `questions` table with the specified columns and rows of data:

id question answer		
(INTEGER) (VARCHAR) PRIMARY KEY NOT NULL		
101 What was Deep Thought's answer?		
102 Where is Wally?		

What will be the result of executing the following code (assuming that `jdbcUrl`, `username`, and `password` are correctly initialized)?

Click here to view code image

```
String findQuestion = "select * from questions where id = ?";
int id = 103;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
     PreparedStatement ps = c.prepareStatement(findQuestion)) {
  ps.setInt(1, id);                                    // (1)
  try (ResultSet rs = ps.executeQuery()) {
    if (rs.next()) {                                   // (2)
      String question = rs.getObject(2, String.class);
      System.out.println(question);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

Select the one correct answer.

a. The program will print one line.
b. The program will print nothing.
c. The program will throw an exception at (1).
d. The program will throw an exception at (2).

24.5 Assume that a database has the following `questions` table with the specified columns and rows of data:

Which of the following statements is true about executing the following code (assuming that `jdbcUrl`, `username`, and `password` are correctly initialized)?

id	question	answer
(INTEGER) PRIMARY	(VARCHAR) NOT NULL	(VARCHAR) NULL
101	What was Deep Thought’s answer?	
102	Where is Wally?	

Click here to view code image

```
String findQuestion = "select question from questions where id = ?";
String provideAnswer = "update questions set answer = ? where id = ?";
int id = 102;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
     PreparedStatement ps1 = c.prepareStatement(findQuestion);
     PreparedStatement ps2 = c.prepareStatement(provideAnswer)) {
  c.setAutoCommit(false);
  ps2.setString(1, "Look and you will find!");
  ps2.setInt(2, id);
  ps2.executeUpdate();
  ps1.setInt(2, id);
  try (ResultSet rs = ps1.executeQuery()) {
    while (rs.next()) {
      String question = rs.getString(1);
      String answer = rs.getString(2);
      System.out.println(question + " - " + answer);
    }
  }
  c.commit();
} catch (SQLException e) {
  e.printStackTrace();
}
```

Select the one correct answer.

a. The program updates one row, executes the query, and commits the transaction.

b. The program updates one row, throws a `SQLException`, and commits the transaction.

c. The program updates one row, throws a `SQLException`, and rolls back the transaction.

d. The program throws a `SQLException`, executes the query, and commits the transaction.

e. The program throws a `SQLException`, executes the query, and rolls back the transaction.

24.6 Which of the following statements is true about the following code (assuming that `jdbcUrl`, `username`, `password`, and `sqlQuery` are correctly initialized)?

Click here to view code image

```
try (Connection connection
         = DriverManager.getConnection(jdbcUrl, username, password);
```

```
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(sqlQuery)) {
  while (resultSet.next()) {
    /* Do nothing. */
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Select the one correct answer.

- **a.** The program will attempt to close the connection, statement, and result set objects, in that exact order.
- **b.** The program will attempt to close the result set, statement, and connection objects, in that exact order.
- **c.** The program will attempt to close the statement, result set, and connection objects, in that exact order.
- **d.** The program will attempt to close the statement, connection, and result set objects, in that exact order.
- **e.** The program will attempt to close the connection, statement, and result set objects, in an undetermined order.

24.7 Which of the following statements is true?

Select the one correct answer.

- **a.** The method `executeQuery()` of the `PreparedStatement` interface accepts marker parameter values.
- **b.** Each `PreparedStatement` object represents a single SQL statement.
- **c.** Each `PreparedStatement` object represents one or more SQL statements.
- **d.** Setting new values for marker parameters in a `PreparedStatement` object will produce a new SQL statement.

24.8 Given the following code:

Click here to view code image

```
String findQuestion
    = "select question, answer from questions where answer is null";
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
     PreparedStatement ps = c.prepareStatement(findQuestion,
         ResultSet.TYPE_FORWARD_ONLY,
         ResultSet.CONCUR_UPDATABLE,
         ResultSet.CLOSE_CURSORS_AT_COMMIT)) {
  c.setAutoCommit(false);
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      rs.updateString("answer", "no answer");
      rs.updateRow();
    }
  }
```

```
    c.commit();
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

Which of the following statements are true about this code (assuming the database supports all designated `ResultSet` features)?
Select the two correct answers.

a. It selects all rows from the `questions` table that have no value set for the `answer` column.

b. It updates all rows in the `questions` table that have no value set for the `answer` column.

c. It sets the `"no answer"` value for the `answer` column in all rows in the `questions` table.

d. It will roll back changes if an exception is thrown inside the `while` loop.

24.9 Given the following code:

Click here to view code image

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
    var ps = c.prepareStatement(
                   "update questions set answer = ? where question = ?")
  ps.setString(2, "What?");                           // (1)
  try (ResultSet rs = ps.executeQuery()) {            // (2)
    while (rs.next()) {                               // (3)
      rs.setString(1, "42");                          // (4)
    }
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Which of the following statements is true about this program?
Select the one correct answer.

a. It updates rows in the `questions` table, setting the `answer` column value to `"42"`.

b. It throws an exception at (1).

c. It throws an exception at (2).

d. It throws an exception at (3).

e. It throws an exception at (4).

24.10 Given the following code:

Click here to view code image

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
```

```
try (Connection c = DriverManager.getConnection(jdbcUrl, username, passw
        PreparedStatement ps = c.prepareStatement(
            "update questions set answer = ? where question = ?")) {
      ps.setString(2,"What?");            // (1)
      ps.setString(1,"42");               // (2)
      ps.execute();                       // (3)
    } catch (SQLException e) {
      e.printStackTrace();
    }
```

Which of the following statements is true about this program? Select the one correct answer.

a. It updates rows in the `questions` table, setting the `answer` column value to `"42"`.

b. It throws an exception at (1).

c. It throws an exception at (2).

d. It throws an exception at (3).

24.11 Which of the following statements are true? Select the three correct answers.

a. A given `PreparedStatement` can only be executed once.

b. A given `PreparedStatement` can be executed multiple times.

c. A given `PreparedStatement` can be executed using both `execute()` and `execute-Query()` methods.

d. A given `PreparedStatement` can be executed using both `execute()` and `execute-Update()` methods.

e. A given `PreparedStatement` can be executed using both `executeUpdate()` and `executeQuery()` methods.

24.12 Which of the following statements is true?

Select the one correct answer.

a. All databases support forward-only type of result sets.

b. Forward-only result sets automatically reflect database changes in an open result set.

c. By default, result set objects are scroll sensitive.

d. Result sets are always closed at commit.

24.13 Which of the following statements is true about `ResultSet` methods?

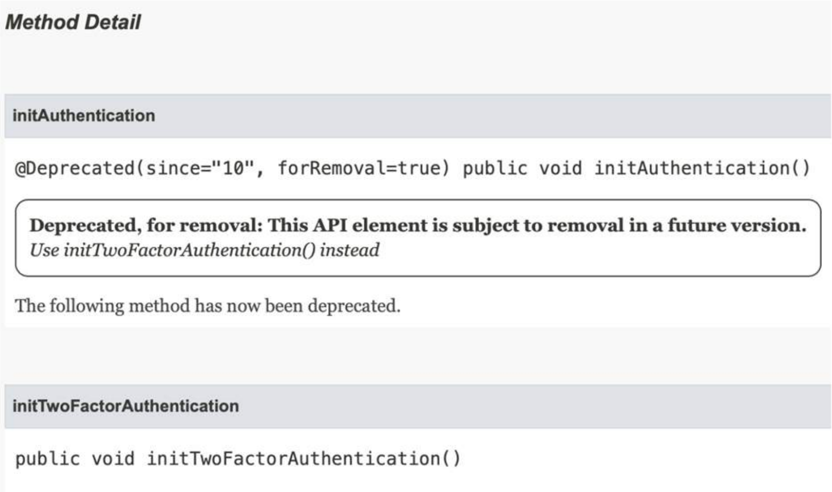
Select the one correct answer.

a. Calling the method `relative(1)` is equivalent to calling the method `next()`.

b. Calling the method `relative(1)` is equivalent to calling the method `absolute(0)`.

c. Calling the method `relative(0)` is equivalent to calling the method `previous()`.

d. Calling the method `relative(-1)` is equivalent to calling the method `absolute(-1)`.



Method Detail

initAuthentication

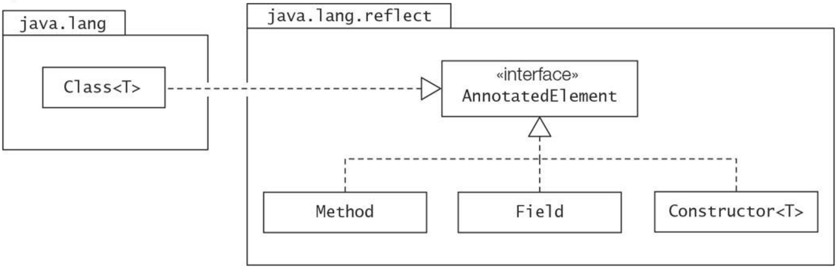
`@Deprecated(since="10", forRemoval=true) public void initAuthentication()`

Deprecated, for removal: This API element is subject to removal in a future version.
Use initTwoFactorAuthentication() instead

The following method has now been deprecated.

initTwoFactorAuthentication

`public void initTwoFactorAuthentication()`


java.lang
Class<T>
java.lang.reflect
«interface»
AnnotatedElement
Method
Field
Constructor<T>

Annotations 25

Chapter Topics

- Understanding the role of annotations to specify metadata in the source code
- Declaring annotation types: marker annotation type, single-element annotation type, and multi-element annotation type
- Understanding the implications of declaring a `value()` element and default values in annotation types

- Applying annotations as instantiations of annotation type declarations with notations for normal, marker, and single-element annotations
- Understanding how the `value()` element and default values in an annotation type declaration influence its application in the source code
- Applying predefined meta-annotations in annotation type declarations to specify the retention (`@Retention`), the target (`@Target`), the inheritance (`@Inherited`), the documentation (`@Documented`), and the repeatability (`@Repeatable`) of an annotation type declaration
- Writing code to demonstrate use of standard predefined annotations to validate overriding (`@Override`) and functional interfaces (`@FunctionalInterface`), annotate deprecated code (`@Deprecated`), and suppress warnings (`@SuppressWarnings`, `@SafeVarargs`)
- Writing code to use the Reflection API to process annotations in a program at runtime

Java SE 17 Developer Supplen	
[22.2] Use Annotations such as Override, FunctionalInterface, Deprecated, SuppressWarnings, and SafeVarargs. • *Note that this is a supplementary objective in the description of the Java SE 17 Developer exam.*	§25.5, p. 1577

Java SE 11 Developer Exam O	
[23.1] Create, apply, and process annotations	§25.1, p. 1557 §25.6, p. 1587

Annotations allow information in the form of metadata to be added to the source code. Annotations provide information about the program, but are not considered to be a part of the program. Syntactically, annotations are a special kind of interfaces that are attached to various program elements in the source code, such as classes, variables, and methods. Tools can then process these annotations at the source level or process the class files into which they are compiled. Without such processing, the annotations do not actually do anything. The benefit of using annotations is only realized when appropriate tools are used to process them and analyze the metadata they represent. There are some existing annotations that are processed by the Java compiler, and even more annotation support is provided by various frameworks like Java EE/Jakarta and the Spring Framework.

It is important to note that annotations are optional and do not change the logic of the program. Augmenting the source code with annotations does not change the bytecode generated for the program. Thus the presence of annotations has no effect on the execution of the program by the JVM, only that they may be present in the class file.

This chapter covers declaring annotation types and applying annotations and utilizing the predefined annotations provided by the Java SE platform. It also provides a taste of processing annotations using the Reflection API.

25.1 Basics of Annotations

Support for annotations is provided by the Annotation API, which is located in the `java.lang.annotation` package. The contents of this package are used to declare annotation types—as we shall see in this chapter.

As subpackages are not automatically imported, the types in the subpackage `java.lang.annotation` of the `java.lang` package must be explicitly imported in order to access them by their simple names—for example, by including a type-import-on-demand statement:

Click here to view code image

```
import java.lang.annotation.*;
```

Annotations are types that are a special kind of interfaces, analogous to enum types that are a special kind of class. An annotation must be declared before it can be used: An *annotation type declaration* must be defined, and when *applied* to a program element, an *annotation* designates a specific invocation of an annotation type. The `java.lang.annotation.Annotation` interface is a common interface implicitly extend by all annotation types, analogous to all classes implicitly extending the `Object` class. However, an annotation type declaration cannot explicitly extend the `Annotation` interface. Typically, an annotation type is declared as a top-level type, but can be nested like static member types. An annotation type declaration is compiled as any other Java type declaration, resulting in a class file containing its bytecode.

Click here to view code image

```
// Annotation type declaration with the name Tag.
```

```
@interface Tag {}              // Declaration always specified with
                               // at-sign (@) preceding the keyword interface.
```

```
// Applying annotation @Tag which denotes the annotation type declaration Tag.
@Tag class Gizmo {            // Always applied with at-sign (@) preceding
                              // the annotation type name.
  @Tag void start() {}        // Another invocation of the annotation type Tag
}
```

The code above shows the minimalistic declaration of an annotation type named `Tag` and its application to a class named `Gizmo`. What is important to note is that an annotation type declaration is specified with the at-sign (`@`) preceding the keyword `interface`, and when applied to a program element, the at-sign (`@`) precedes the annotation type name, and each application of the annotation to any program element denotes a different invocation of the annotation type. The rest of this chapter will elaborate on defining annotation type declarations and applying annotations to various program elements in the source code. The last section provides an introduction to *annotation processing*.

25.2 Declaring Annotation Types

An *annotation type declaration* specifies a new *annotation type*, and has a lot in common with an interface type. General syntax of an annotation type declaration is shown below.

Click here to view code image

```
meta-annotations access_modifier @interface annotation_name // Annot type head
{ // Annotation type body with zero or more member declarations:

  annotation_type_element_declarations

  constant_variable_declarations

  enum_declarations
  class_declarations
  interface_declarations
}
```

We will use the annotation type declaration `TaskInfo` below to explain the syntax of an annotation type declaration. The `TaskInfo` annotation type allows meta-information to be specified about tasks on program elements: a description of the task, people the task is assigned to, and the priority given to the task. The `TaskInfo` annotation type is meant to illustrate declaring annotation types, and is in no way meant to replace task management tools.

Click here to view code image

```
import java.lang.annotation.Target;                // (1)
import java.lang.annotation.ElementType;           // (2)
import java.lang.annotation.Retention;             // (3)
import java.lang.annotation.RetentionPolicy;       // (4)

@Target({ElementType.TYPE, ElementType.METHOD})    // (5) Meta-annotation
@Retention(RetentionPolicy.RUNTIME)                // (6) Meta-annotation
public @interface TaskInfo {                       // (7)
  String taskDesc();                               // (8) Annotation type elem
  String[] assignedTo();                           // (9) Annotation type elem
  TaskPriority priority() default TaskPriority.NORMAL;  // (10) Annot type ele

  public enum TaskPriority { LOW, NORMAL, HIGH };       // (11) Nested enum ty

  public static final String LOG_FILE = "./logs/Tasks.log"; // (12) const decl
}
```

The *meta-annotations* specified in the annotation type header are annotations that are applied to the annotation type declaration to specify certain aspects of the declaration—that is, they allow metadata to be associated with the annotation type declaration (p. 1567). In the code above, these meta-annotations are defined by classes in the `java.lang.annotation` package, which are imported at (1) and (3), together with auxiliary enum types at (2) and (4). The meta-annotation `@Target` at (5) specifies which program elements the annotation type can be applied to. The targets specified for the `TaskInfo` annotation type are any type declaration (`Element-Type.TYPE`) and methods (`ElementType.METHOD`). The `@Retention` meta-annotation specifies the retention policy: whether invocations of the annotation should be retained with the program element in the source file or in the class file, or made available at runtime. Applications of the `TaskInfo` annotation type on program elements will be available at runtime (`RetentionPolicy.RUNTIME`).

The *access modifier* that can be specified in the annotation type header is `public`, or there is no access modifier to indicate package accessibility—the same as with a normal interface. The `TaskInfo` annotation type is declared `public`.

Like any normal interface, an annotation type declaration is implicitly abstract, but seldom explicitly specified as such. Note also that an annotation type declaration cannot be generic and the extends clause is not permitted—in contrast to a normal interface.

The *simple name* of an annotation type shares the same namespace as simple names of normal classes and interfaces in a package—that is, a simple name of an annotation type should not conflict with any type declaration having the same simple name in the package.

The *annotation type body* can contain zero or more *member declarations*. These member declarations are analogous to the member declarations in a normal interface when it comes to constant variable declarations (which are implicitly public, static, and final) and nested type member declarations—but default, static, and private methods are not permitted. For example, the `TaskInfo` type annotation declaration declares a nested enum type named `TaskPriority` at (11) and a constant variable named `LOG_FILE` at (12).

In the rest of this section, we take a closer look at declaration of annotation type elements in the body of an annotation type, declared at (8), (9), and (10), as values provided for these annotation type elements when an annotation is applied to a program element determine the metadata that is associated with the program element.

Declaring Annotation Type Elements

The body of an *annotation type declaration* may contain declarations of *annotation type elements*. The syntax of an annotation type element declaration is shown below.

Click here to view code image

```
element_modifiers element_type method_name() default constant_expression;
```

A partial declaration of the `TaskInfo` annotation type is repeated below, showing the declaration of the three annotation type elements at (8), (9), and (10).

Click here to view code image

```
// ...
public @interface TaskInfo {                   // (7)
  String taskDesc();                           // (8) Required annotation elem
  String[] assignedTo();                       // (9) Required annotation elem
  TaskPriority priority() default TaskPriority.NORMAL;   // (10) Optional
                                                         // annotation element
  // ...
}
```

Each annotation type element specifies a *method declaration*, resembling the abstract method declaration in a normal interface. Each annotation type element is usually referred to as an *element* of the annotation type. The elements declared in an annotation type declaration constitute the *attributes* of the annotation type.

The *method declaration* in an element has a *return type* and a *method name*, and is *always* specified with an *empty formal parameter list*, as exemplified by the method declarations in the elements at (8), (9), and (10) in the `TaskInfo` annotation type declaration. Note that no formal parameters, type parameters, or a throws clause can be specified for a method declaration in an element. The name and the return type of the method declared in an element are referred to as *the name and the type of the element*, respectively. Only specific types can be declared as the type of an element (p. 1561).

Annotation types are characterized by the number of elements declared in their declaration.

- *A marker annotation type* is an annotation type with no elements, analogous to a marker interface. A marker annotation is typically used to indicate a specific purpose that does not require any elements. Earlier in this chapter we have seen the annotation type `Tag`, which is a marker annotation type. The notation for applying such an annotation type to program elements is straightforward: an at-sign (`@`) preceding the annotation name (p. 1565)—for example, `@Tag`. Ample examples of marker annotation types can be found throughout this chapter.

  Click here to view code image

  ```
  @interface Tag {} // No element type declarations.
  ```

- *A single-element annotation type*, as the name suggests, is an annotation type with a single element. By convention, the method of a single-element annotation type is declared as `value()`. Various shorthand notations are offered when applying such annotation types to program elements (p. 1565).

  Click here to view code image

  ```
  @interface SecurityLevel {                  // Single-element type declar
     int value();
  }
  ```

- *A multi-element annotation type*, as the name suggests, is an annotation type with more than one element. The `TaskInfo` annotation type is a multi-element annotation type, as it declares three elements. At the most, one method of an element in a multi-element annotation type can be declared as `value()`. Under certain conditions, a shorthand notation can be used when applying an annotation type declaration that has a `value()` element (p. 1563).

Element Modifiers for Annotation Type Elements

The modifiers allowed for an annotation type element are analogous to the modifiers that can be specified for an abstract method declaration in a normal interface. The only *access modifier* that can be specified for an annotation type element is `public`; otherwise, the accessibility is implicitly public. The modifier `abstract` is also implied, and typically not specified in the annotation type element declaration. The elements of the `TaskInfo` annotation type are implicitly `public` and `abstract`.

Element Type of Annotation Type Elements

The *element type* of an annotation type element can be any one of the following:

- A primitive type
- A `String`
- The class `Class`
- An enum type
- An annotation type
- A one-dimensional array whose element type is one of the above types

The code below illustrates declaring the element type of an annotation type element. An example of each type that is allowed as the return type of an annotation type element is shown, together with the specification of a default value for each annotation type element.

Click here to view code image

```
public @interface MultiElementAnnotationType {

  public enum Priority { LOW, NORMAL, HIGH };

  public int certificationLevel()     default 1;                        // int
  String date()                       default "2021-01-11";   // String
  Class<? extends PrettyPrinter> pp()
                        default AdvancedPrettyPrinter.class;          // typ
  Priority priorityLevel()               default Priority.NORMAL;       // enu
  Tag annotate()                         default @Tag;                  // Ann
  int[] value()                          default {10, 20, 30};          // Arr
}
// Auxiliary classes:
class PrettyPrinter {}
class AdvancedPrettyPrinter extends PrettyPrinter {}
```

The declaration of the `ProblematicAnnotationType` annotation type below shows examples of illegal declarations of annotation type elements.

Click here to view code image

```
@interface ProblematicAnnotationType {
  StringBuilder message();              // Illegal return type.
  int[][] voting();                     // Only one-dimensional array allowed.
  String value;                         // Missing parentheses.
  private Thread.State state();         // Only public can be specified.
}
```

The code below shows an annotation which is used as an element type within another annotation. This relationship between annotations is sometimes described as a *contained annotation type* (1) and a *containing annotation type* (2). This relationship is further explored when we discuss repeatable annotations (p. 1575).

Click here to view code image

```
public @interface MusicMeta {                 // (1) Contained annotation type
  String value();
}
public @interface ArtistMeta {                // (2) Containing annotation type
  MusicMeta value();                          // Annotation type from (1).
}
```

Defaults for Annotation Type Elements

An annotation type element declaration can declare a *default value* for an element using the optional `default` clause in the annotation type element declaration, as shown in the declaration of the `MultiElementAnnotationType` earlier. The default value is specified in the `default` clause as a *constant expression*—that is, the operands of the expression are all constants so that the compiler is able to determine its value at compile time. The constant expression must not be `null`. The declaration of the `Refactor` annotation type below shows examples of illegal default values.

Click here to view code image

```
@interface Refactor {
  int id()          default (int) (Math.random() * 10);// Not a const expressi
  String value()    default null;                      // Cannot be null.
  String deadline() default new String("2021-01-11");  // Not a const expressi
  String[] team()   default new String[] {"VJ", "PT"}; // Not a const expressi
}
```

An element that specifies a default value is called an *optional annotation element*. An element that does *not* specify a `default` clause is called a *required annotation element*. As we shall see when an annotation is applied to a program element, specifying a value for an optional annotation element can be omitted, in which case the default value in the declaration of the annotation type element is used. For a required annotation element, a value for the element *must* be specified when the annotation is applied to a program element. We see that the declaration of the `TaskInfo` annotation type below has two required annotation elements, at (8) and at (9), and one optional annotation element at (10).

Click here to view code image

```
// ...
public @interface TaskInfo {                   // (7)
  String taskDesc();                           // (8) Required annotation elem

  String[] assignedTo();                       // (9) Required annotation elem
  TaskPriority priority() default TaskPriority.NORMAL; // (10) Optional
                                                       //      annotation elem
  // ...
}
```

25.3 Applying Annotations

Once an annotation type has been compiled, it can be applied to program elements. An annotation is an application of an annotation type to a program element, and we refer to the annotation as being of that type. Which program elements an annotation can be applied to is typically determined by the meta-annotation `@Target` specified in the declaration of the annotation type (p. 1569). Each annotation refers to a specific invocation of an annotation type. An annotation is always specified with the at-sign (`@`) preceding the name of the annotation type, and usually provides values for the elements of the annotation type.

Multiple annotations of different types can be applied to a program element. Repeatable annotations are covered later (p. 1575).

Annotations can be characterized according to the *notation* that can be used when applied to a program element. We distinguish between three kinds of annotations:

- *Normal annotations*
- *Marker annotations*
- *Single-element annotations*

The last two annotations are special cases of normal annotations. Choosing which annotation to use depends on knowing what annotation type elements are declared in an annotation type declaration. The different kinds of annotations are discussed in this section.

Applying Normal Annotations

When a *normal annotation* is applied to a program element, in addition to the name of the annotation type being preceded by the at-sign (`@`), it can optionally specify a comma-separated list of element-value pairs enclosed in parentheses, `()`. Each pair has the syntax *element* `=` *value*. For each pair, the *element* and the *value* must be compatible with the method name and the return type of a method specified in an annotation type element.

The normal annotation for the annotation type `TaskInfo` (p. 1558) is applied to a class below, where each element-value pair associates a value with an element of the annotation type `TaskInfo`.

Click here to view code image

```
@TaskInfo(                                                 // Normal annotati
   taskDesc = "Class for monitoring nuclear reactor activity",      // Requir
   assignedTo = {"Tom", "Dick", "Harriet"},                          // Requir
   priority = TaskInfo.TaskPriority.HIGH                             // Option
   )
class NuclearPlant {}
```

A normal annotation must contain element-value pairs for all required annotation elements in the annotation type—that is, for elements that do not specify a default value. Specifying the element-value pairs for optional annotation elements is obviously optional.

The order of the element-value pairs is irrelevant, but usually they are specified in the same order that the elements have in the annotation type declaration. Note also that a `null` value in an element-value pair will result in a compile-time error.

In the normal annotation below, the optional annotation element `priority` in the annotation type `TaskInfo` is omitted.

Click here to view code image

```
@TaskInfo(                                                          // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",      // Requir
   assignedTo = {"Tom", "Dick", "Harriet"}                           // Requir
   )
class NuclearPlant {}
```

The above normal annotation is then equivalent to the normal annotation below, where the default value `TaskInfo.TaskPriority.NORMAL` of the optional annotation element `priority` is implied.

Click here to view code image

```
@TaskInfo(                                                                // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",            // Requir
   assignedTo = {"Tom", "Dick", "Harriet"},                               // Requir
   priority = TaskInfo.TaskPriority.NORMAL                                // Implie
   )
class NuclearPlant {}
```

An *array element initializer*, `{v1, ..., vn}`, can be used to specify the values of array elements for annotation elements whose type is an array type. The curly brackets can be omitted when specifying the value for a *single-element array-valued element-value pair*—that is, where the type of the annotation type element is an array type, but only a single array element is specified in the annotation.

Click here to view code image

```
@TaskInfo(
    priority = TaskInfo.TaskPriority.LOW,
    taskDesc = "Start nuclear reactor",
    assignedTo = "Harriet"                // Single-element array-valued eleme
    )
class NuclearPlant {}
```

Applying Marker Annotations

A *marker annotation* is a shorthand notation typically for use with marker annotation types, which have no elements in their type declaration.

Click here to view code image

```
// Marker annotation type
@interface Tag {}                       // No element type declarations
```

The marker annotation of annotation type `Tag` is applied to a class as shown below, with just the name of the annotation type preceded by the at-sign (`@`)—that is, the parentheses are omitted.

Click here to view code image

```
@Tag class Gizmo {}                     // Marker annotation
```

The normal annotation of annotation type `Tag` is applied to a class, where the empty list of element-value pairs is explicitly specified:

Click here to view code image

```
@Tag() class Gizmo {}                   // Normal annotation
```

A marker annotation can also be used if all elements are optional annotation elements in the annotation type declaration—that is, they specify default values.

Click here to view code image

```
// Annotation type declaration where all elements have default values.
@interface Option {
  Color color()   default Color.WHITE;   // Optional annotation element
  Size size()     default Size.M;        // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

A marker annotation of annotation type `Option` is applied to the class below:

Click here to view code image

```
@Option                                                    // Marker annotatio
class Item {}
```

The above marker annotation is equivalent to the following normal annotation:

Click here to view code image

```
@Option(color = Option.Color.WHITE, size = Option.Size.M)  // Normal annotatio
class Item {}
```

Applying Single-Element Annotations

A *single-element annotation* is a shorthand notation typically for use with single-element annotation types, whose only annotation type element is declared as a `value()` element. Single-element annotation will not work if the element in the single-element annotation type is not declared as a `value()` element. However, there is no requirement that the single element should be either required or optional. Note that there is no restriction on declaring a `value()` element for any annotation type, but that alone does not guarantee that single-element annotation can be used.

Click here to view code image

```
// Single-element annotation type
```

```
@interface Author {
  String value();                               // Single annotation type element
}
```

A single-element annotation of annotation type `Author` is applied to the class below, where only the value of the single `value()` element is specified:

Click here to view code image

```
@Author("Tom")                                  // Single-element annotation
class Connection {}
```

The above single-element annotation is equivalent to the following normal annotation:

Click here to view code image

```
@Author(value = "Tom")                          // Normal annotation
class Connection {}
```

Single-element annotation can also be used if one element is declared as a `value()` element and *all other elements are optional annotation elements* in the annotation type declaration—that is, they specify default values.

Click here to view code image

```
//Annotation type declaration with a value() element and other elements with
default values.
@interface ExtraOption {
  int value();                                  // Required annotation element
  Color color()   default Color.WHITE;          // Optional annotation element
  Size size()     default Size.M;               // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

A single-element annotation of annotation type `ExtraOption` is applied to the class below:

Click here to view code image

```
@ExtraOption(10)                                      // Single-element annotatio
class ItemV2 {}
```

The above single-element annotation is equivalent to the following normal annotation:

Click here to view code image

```
@ExtraOption(value=10, color=ExtraOption.Color.WHITE,       // Normal annotatio
             size=ExtraOption.Size.M)
class ItemV3 {}
```

Following is an example of using *array-valued single-element annotation*—that is, where the element type of the single annotation type element is an array type:

Click here to view code image

```
// Array-valued single-element annotation type declaration:
@interface Problems {
  String[] value();                              // Array-type value() eleme
}
@Problems({"Code smell", "Exception not caught"})  // Array-valued single-elem
class ItemV4 {}                                  // annotation

@Problems(value = {"Code smell", "Exception not caught"})  // Normal annotatio
class ItemV5 {}
```

Using a *single-element array-valued single-element annotation*—that is, where the element type of the single `value()` element is an array type, but only a single array element is specified in the annotation—is shown below.

Click here to view code image

```
@Problems("Code smell")   // Single-element array-valued single-element annota
class ItemV6 {}

@Problems({"Code smell"}) // Single-element array-valued single-element annota
class ItemV7 {}

@Problems(value = "Code smell")      // Normal annotation
class ItemV8 {}

@Problems(value = {"Code smell"})    // Normal annotation
class ItemV9 {}
```

25.4 Meta-Annotations

Meta-annotations are annotations that can be applied on an annotation type declaration to specify various aspects of the annotation when this annotation type is applied in the source code. We discuss the purpose and usage of predefined meta-annotations when declaring an annotation type, as they have implications on how the annotation will be handled when applied in the source code.

`@Retention`: Specifies how this annotation is retained when applied, whether in the source file, in the class file, or at runtime.

`@Target`: Specifies to which contexts in the source code this annotation can be applied.

`@Inherited`: Indicates that all subclasses of a superclass that this annotation is applied to will inherit this annotation.

`@Repeatable`: Indicates that this annotation can be used multiple times in the same context.

`@Documented`: Indicates that this annotation should be included in the documentation generated by a tool like `javadoc`.

The `@Retention` Meta-Annotation

In the declaration of an annotation type, it is possible to specify how long the annotation will be retained, meaning whether it should be in the source file, in the class file, or available at runtime.

In fact, each annotation is associated with one of three retention policies, controlled by the meta-annotation type `java.lang.annotation.Retention` and the *retention policy constants* defined by the enum type `java.lang.annotation.RetentionPolicy` shown in Table 25.1 in increasing order of retention.

The meta-annotation type `Retention` is a *single-element meta-annotation type* that specifies a `value()` element of enum type `RetentionPolicy`. Defining `value()` elements is covered elsewhere in this chapter (p. 1560).

Click here to view code image

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value = RetentionPolicy.RUNTIME)    // Retention: In source code,
                                               // in class file, at runtime.
public @interface MusicMeta {}
```

The annotation type `MusicMeta` above needs to be retained at runtime so that it is possible to extract any information that is specified by this annotation. The `Retention-Policy.RUNTIME` enum constant provides the longest retention. If used in the source code, the `@MusicMeta` annotation will be recorded in the class file, and when loaded at runtime, it will be available through reflection. However, no provision has been made as yet in the `MusicMeta` annotation type declaration to provide any information. It is added to the annotation type declaration by specifying annotation type elements (p. 1560).

Click here to view code image

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value = RetentionPolicy.SOURCE) // Retention: Only in the source co
public @interface Override {}
```

The code above shows the declaration of the `Override` annotation type. This builtin annotation type when applied in the source code can be used by the compiler (p. 1578). Its declaration specifies the `RetentionPolicy.SOURCE`, meaning that the annotation will only be retained in the source code. In other words, it will not be recorded in the class file or be available at runtime.

The annotation type declarations shown above do not contain any annotation type elements. They are *marker annotation types*. The purpose of a marker annotation type is to indicate the presence of a feature, rather than convey any specific information. As the `MusicMeta` annotation type declaration stands, it is possible to discover at runtime that this annotation is used on a particular program element and take the appropriate action, whatever that might be. Use of the `@Override` marker annotation in the source code alerts the compiler to check that the criteria for overriding is satisfied by the method on which this annotation is applied, but the annotation is not recorded in the class file.

Table 25.1 *Retention Policy Values for the* `@Retention` *Meta-Annotation*

Constants defined by the `Rete`	**Description**
`SOURCE`	This retention policy implies that the annotation is only retained in the source code and is discarded by the compiler—that is, it is not recorded in the class file. The purpose of this annotation is to provide information for the compiler—for example, to validate source code, detect design errors, and suppress compiler warnings.
`CLASS`	This retention policy implies that the annotation is recorded in the class file and thus retained by the compiler, but it is not intended to be used at runtime. That it, it need not be loaded by the JVM at runtime. The purpose of this annotation is to provide information for compile-time and deployment-time processing. It can be used by tools that process information to generate additional code—for example, deployment descriptors such as XML files used by Java EE/Jakarta. `RetentionPolicy.CLASS` is the default retention policy if no retention policy is specified on the annotation type declaration.
`RUNTIME`	This retention policy results in the annotation being recorded in the class file by the compiler. It is retained by the JVM and can be read at runtime via reflection.

The `@Target` Meta-Annotation

An annotation type declaration can specify the contexts in which the annotation is applicable. This is controlled by the meta-annotation type `java.lang.annotation.Target` and the *element type constants* defined by the enum type `java.lang.annotation.Element-Type`, shown in Table 25.2. The meta-annotation type `Target` is an *array-valued single-element meta-annotation type*—that is, it has a single `value()` element of type `ElementType[]` to specify contexts in which the annotation is applicable, meaning multiple `ElementType` values can be specified to indicate multiple contexts.

Table 25.2 *The* `ElementType` *Values for the* `@Target` *Meta-Annotation*

Constants defined by the `ElementType` en	**Description**
`ANNOTATION_TYPE`	The annotation can be applied to annotation type declarations. `@Tag @interface Status {}`
`CONSTRUCTOR`	The annotation can be applied to constructor declarations. `@Tag public Item() {}`

`FIELD`	The annotation can be applied to field declarations, such as instance and static variables, or enum constants.
	Click here to view code image
	`@Tag public static final int YEAR = 2021;`
	`enum Size { @Tag S, @Tag M, @Tag L, @Tag XL }`

`LOCAL_VARIABLE`	The annotation can be applied to local variable declarations, including loop variables of `for` statements and resource variables of `try`-with-resources statements.
	Click here to view code image
	`@Tag String localBarber = "Director's Cut";`
	`for (@Tag int i : intArray) {}`
	`try (@Tag var resource = new CloseableResource()) {}`

`METHOD`	The annotation can be applied to method declarations (including elements of annotation types).
	Click here to view code image
	`@Tag boolean passed() { return true; }`
	`@Tag String[] value();`

`MODULE`	The annotation can be applied to a module declaration in a `module-info.java` source file. `@Tag module com.passion {}`
`PACKAGE`	The annotation can be applied to a package declaration in a source file. Best practice is to specify package-related annotations in a file named `package-info.java`. Click here to view code image `@Tag package com.passion.logic;`
`PARAMETER`	The annotation can be applied to formal and exception parameter declarations in constructors, methods, lambda expressions, and `catch` blocks. Click here to view code image `double circleArea(@Tag double r) { return Math.PI*r*r; }`

```
Predicate<String> p = (@Tag var str) -> str.length() < 10;
try {} catch (@Tag Exception ex) {}
```

9 `TYPE`

The annotation can be applied to type declarations: class, interface (including annotation type), and enum declarations.

Click here to view code image

```
@Tag class Gizmo {}
@Tag interface Reparable {}
@Tag enum Direction {LEFT, RIGHT}
@Tag @interface Validate {}
```

10 `TYPE_PARAMETER`

The annotation can be applied to type parameter declarations of generic classes, interfaces, methods, and constructors.

Click here to view code image

```
class Box<@Tag E> {}
interface Eatable<@Tag E> {}
<@Tag E extends Comparable<E>> void comparison(E e) {}
<@Tag E> Item(E e) {}
```

11 `TYPE_USE`

The annotation can be applied in any type context in declarations and expressions where a type can be used.

Click here to view code image

```
class X extends @Tag Y implements @Tag IZ {}
java.lang. @Tag String strOp() { return "ok"; }
@Tag int compute() { return (@Tag int) 3.14; }
java.lang. @Tag Integer iRef1 = 100;
@Tag Integer iRef2 = 100;
void passItOn() throws @Tag Exception {}
@Tag int[] array1;         // Annotates primitive type int
int @Tag [] array2;        // Annotates array type int[]
int @Tag [][] array3;      // Annotates array type int[][]
```

The target values of the `ElementType` constants in Table 25.2 are categorized by the two contexts where annotations can be applied: *declaration context* and *type context*.

The target values of the `ElementType` constants from 1 to 10 in Table 25.2 indicate *declaration contexts*—that is, a context where a program element is declared. Annotations with these target values are called *declaration annotations*. For example, an annotation type declaration with the target value `ElementType.CONSTRUCTOR` is only applicable to a constructor declaration.

Click here to view code image

```
@Target(ElementType.CONSTRUCTOR)
@interface Tag {}
______________________________
class Item {
  @Tag public Item() {}       // Declaration context. Annotates constructor It
}
```

However, an annotation type declaration with the target value `ElementType.PARAMETER` is applicable to a parameter declaration in contexts where such parameter declarations are permitted. Note that several target values can be specified in the `@Target` meta-annotation.

Click here to view code image

```
@Target({ElementType.CONSTRUCTOR, ElementType.PARAMETER})
@interface Tag {}
_________________________________
double circleArea(@Tag double r) { // Declaration context. Annotates parameter
  return 5.0;
}
```

Note also that the context of the target value `ElementType.ANNOTATION_TYPE` and the context of the target value `ElementType.TYPE` are overlapping, as the latter not only includes annotation type declarations, but also other reference type declarations (class, interface, and enum).

The target value `ElementType.TYPE_USE` indicates *type contexts*—that is, where a type is used in a declaration or an expression. Annotations with this target value are called *type annotations*.

Click here to view code image

```
@Target(ElementType.TYPE_USE)
@interface Tag {}
_________________________________
java.lang. @Tag Integer iRef1 = 100;          // (1) Type context. Annotates Int
@Tag Integer iRef2 = 100;                     // (2) Type context. Annotates Int
@Tag java.lang.Integer iRef3 = 100;          // (3) Declaration context.
                                                    Compile-time error!
@Tag int compute() { return (@Tag int) 3.14; }// (4) Type context is int type.
@Tag void compute(int i) {}                     // (5) No return type specified.
                                                //     Compile-time error!
```

In both (1) and (2) above, `@Tag` is a type annotation on the type `Integer`, as this type is used in the declaration of the variables. The code at (3) does not compile, as the type annotation `@Tag` is used in a declaration context. Note how the position of the annotation changes the context of the annotation at (1) and (3). At (4), the type annotation `@Tag` is on the return type `int` of the non-`void` method. As the method at (4) is a `void` method, there is no return type that is applicable for the type annotation `@Tag`, so the compiler flags an error.

Consider the following annotation type with the target value `ElementType.FIELD`:

Click here to view code image

```
@Target(ElementType.FIELD)
@interface Tag {}
__________________________________
java.lang. @Tag Integer iRef1 = 100; // (1a) Type context. Compile-time error!
@Tag Integer iRef2 = 100;               // (2a) Declaration context. Annotates
```

```
@Tag java.lang.Integer iRef3 = 100;   // (3a) Declaration context. Annotates i
```

At both (2a) and (3a), the annotation `@Tag` is a declaration annotation on the field, as these are field declarations. The code at (1a) does not compile, as the declaration annotation `@Tag` is used in a type context.

An annotation can be both a declaration and a type annotation, as illustrated by the code below.

Click here to view code image

```
@Target({ElementType.TYPE_USE, ElementType.FIELD})
@interface Tag {}
__________________________________

java.lang. @Tag Integer iRef1 = 100; // (1b) Type context. Annotates Integer.
@Tag Integer iRef2 = 100;            // (2b) Declaration and type context.
@Tag java.lang.Integer iRef3 = 100;   // (3b) Declaration context. Annotates i
```

An `ElementType` value can only occur once in the `value()` element of the meta-annotation `@Target`. The following specification of the `value()` element of the meta-annotation `@Target` will not compile.

Click here to view code image

```
@Target(value = {ElementType.METHOD, ElementType.FIELD,
                 ElementType.METHOD})       // Compile-time error!
@interface Tag {}
```

By default (when the `@Target` meta-annotation is not present on the annotation type declaration), the annotation can be applied in any declaration context *except type parameter declarations*. In addition, the annotation is *not permitted in any type contexts*.

Click here to view code image

```
@interface Tag {}               // No target specified.
__________________________________________________________________________

@Tag class Box<@Tag E> {        // Class declaration: OK.
                                // Type parameter: Compile-time error!
  <@Tag T> void doIt(T t) {}    // Type parameter: Compile-time error!
  void doTask() throws @Tag Exception {}     // Type context: Compile-time err
  int iTod(double d) { return (@Tag int) d; }// Type context: Compile-time err
  @Tag boolean flag;                         // Field declaration. OK.
}
```

An empty `@Target(value = {})` annotation is allowed, shown at (1) below, but it does not allow the annotation to be directly applied to any program element, as shown at (3) below. However, this annotation can still be used as the type of an element in other annotation type declarations, as shown at (2), which can be applied normally, as shown at (4).

Click here to view code image

```
@Target({})                                 // (1) Empty target.
@interface Exclusive {}

@interface ExtraExclusive {
  Exclusive value() default @Exclusive;     // (2) Annotation as element type.
}
_____________________________________________________________________________
@Exclusive class Titanium {}                // (3) Compile-time error!
@ExtraExclusive class Krypton {}            // (4) OK.
```

The annotation type declaration below shows that the `MusicMeta` type annotation is applicable to any type (class, interface, and enum), method, or field.

Click here to view code image

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(value = {ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
public @interface MusicMeta {}
```

The `@MusicMeta` annotation is applied below on a class at (1), on a field at (2), and on a method at (3), as permitted by the annotation type declaration shown above. Not surprisingly, applying it on a local variable at (4) results in a compile-time error.

Click here to view code image

```
@MusicMeta public class Composition {         // (1) On class
  @MusicMeta private String description;      // (2) On field
  @MusicMeta public void play() {             // (3) On method
//  @MusicMeta int volume;                    // (4) Compile-time error!
  }
}
```

The `@Inherited` Meta-Annotation

By default, an annotation applied on a class is not inherited by its subclasses. However, this can be changed by applying the meta-annotation `@Inherited` on the declaration of the annotation type. Such an annotation will automatically be inherited by subclasses when this annotation is present in their superclass. It has no effect if the annotation is applied to program elements other than classes. Note also that a class cannot inherit annotations from interfaces it implements, nor can a subinterface inherit annotations from superinterfaces it extends, regardless of the fact that these annotations are marked with the meta-annotation `@Inherited`.

The meta-annotation `@Inherited` is of type `java.lang.annotation.Inherited`, which is a *marker meta-annotation type*.

Click here to view code image

```
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Playable {}
```

```
@MusicMeta
@Playable
public class Composition {}
```

```
public class Song extends Composition {}
```

In the code above, the annotations `@MusicMeta` and `@Playable` are applied to the class `Composition`, but only the annotation `@Playable` is inherited by the subclass `Song` because only the `@Playable` annotation is applied to the meta-annotation `@Inherited` in its type declaration. This means that the `Composition` class has two annotations, but the subclass `Song` has only one, namely the inherited annotation `@Playable`.

The `@Documented` Meta-Annotation

By default, annotations are not processed by the `javadoc` tool when generating API documentation from the source code, unless the declaration of the annotation type is marked with the meta-annotation `@Documented`. This meta-annotation is of type `java.lang.annotation.Documented` which is a *marker meta-annotation type*.

Click here to view code image

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented                             // (1)
public @interface Pending {}
```

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Verified {}          // (2)
```

```
@Documented
@Verified
```

```
public class Gizmo {}                   // (3)
```

In the code above, the declaration of the annotation type `Pending` is marked with the meta-annotation `Documented` at (1), but this is not the case for the declaration of the annotation type `Verified` at (2). Both annotations are applied to the class `Gizmo` at (3). As a result, the generated Javadoc documentation for the class `Gizmo` will only show the presence of the `@Pending` annotation, but not the `@Verified` annotation. Thus the `@Documented` annotation gives annotation developers full control over the process of including annotations in the generated API documentation.

For another example of using the meta-annotation `@Documented`, see the discussion about the `@Deprecated` annotation (Example 25.1, p. 1580).

The `@Repeatable` Meta-Annotation

There are situations where we would like to apply the same annotation multiple times on the same program element. This makes more sense if the annotation is not a marker annotation and we want to apply the annotation with different values.

The declaration of the `Choice` annotation type below declares two annotation type elements that specify the color and the size.

Click here to view code image

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

We wish to apply the `@Choice` annotation multiple times to the class `Item` to indicate different combinations of color and size. However, the code below will not compile.

Click here to view code image

```
@Choice(color="Green", size=Size.S)                     // Compile-time error!
@Choice(color="Yellow", size=Size.XL)                   // Compile-time error!
@Choice(color="Red", size=Size.M)                       // Compile-time error!
@Choice(color="White")                                  // Compile-time error!
class Item {}
```

The compiler complains that the `@Choice` annotation is not *repeatable*. The declaration of the *Choice* annotation type must be declared with the meta-annotation `@Repeatable`, which is a *single-element meta-annotation* having the following declaration:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface Repeatable {
  Class<? extends Annotation> value();  // Indicates the containing annot type
}
```

In order to apply the `@Repeatable` meta-annotation on an annotation type, the value of the *containing annotation type* must be supplied—that is, the `Class` object representing the containing annotation type in its single-element type declaration must be specified. Keep in mind that all annotation types implement the `Annotation` interface.

We can augment the declaration of the `Choice` annotation with the meta-annotation `@Repeatable` that specifies the containing annotation type `Choices` (which has not been declared as yet).

Click here to view code image

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Choices.class)    // @Repeatable specifies the container
                              // annotation type Choices.
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

The *containing annotation type* (`Choices`) of a *repeatable annotation type* (`Choice`) must declare a `value()` method whose return type is *an array of the repeatable annotation type* (`Choice[]`).

Click here to view code image

```
@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  Choice[] value();              // Return value is an array of Choice.
}
```

The upshot of the setup described above is that the multiple applications of a *repeatable annotation* (`@Choice`) will be stored in the *array of the repeatable annotation type* (`Choice[]`) declared in the *containing annotation type* (`@Choices`).

The multiple applications of the `@Choice` annotation on the class `Item` above will now compile. We can call the method

`AnnotationPrinter.printAllAnnotations()` in Example 25.5, p. 1592, with the `Class<Item>` object `Item.class` to print the annotations on the class `Item`. (*Output shown below has been edited to fit on the page.*)

Click here to view code image

```
Annotations for 'class Item':
  @Choices(value={@Choice(size=S, color="Green"),
                  @Choice(size=XL, color="Yellow"),
                  @Choice(size=M, color="Red"),
                  @Choice(size=L, color="White")})
...
```

A *Containing Annotation Type* (*CAT*) for a *Repeatable Annotation Type* (*RAT*) must satisfy the following conditions:

- *RAT* must specify *CAT*`.class` as the value of the single element of its meta-annotation `@Repeatable`.
- *CAT* must declare a `value()` annotation type element whose type is *RAT*`[]`.
- Any additional annotation type elements declared in *CAT* must have a default value.

  Click here to view code image

  ```
  @Retention(RetentionPolicy.RUNTIME)
  @interface Choices {
    Choice[] value();                  // Return value is an array of Ch
    double minPrice() default 1.00;    // Must specify a default value.
  }
  ```

- The retention policy for *CAT* is at least as long as the retention policy for *RAT*. For example, if *RAT* has retention policy `RUNTIME`, the retention policy for *CAT* cannot be `CLASS` or `SOURCE`. However, if *RAT* has retention policy `CLASS`, the retention policy for *CAT* can be `CLASS` or `RUNTIME`.
- *RAT* is applicable to at least the same kind of program elements as *CAT*—that is, the list of targets to which *RAT* can be applied cannot be shorter than the list of targets to which *CAT* can be applied. For example, if *CAT* is applicable to the targets `METHOD` and `FIELD`, then *RAT* must be applicable to at least these targets.
- If *RAT* has the meta-annotation `@Documented`, then so must

CAT.

- If *RAT* has the meta-annotation `@Inherited`, then so must *CAT*.

In addition, it is a compile-time error if the meta-annotation `@Repeatable` in the repeatable annotation type does not specify a containing annotation type for the repeatable annotation type.

Click here to view code image

```
@Repeatable(Choices.class)        // Compile-time error since invalid return t
                                  // at (1).
@interface Choice {
  String color();

  Size size() default Size.L;
}

@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  String[] value();                // (1) Invalid return type.
}
```

25.5 Selected Standard Annotations

The Java SE Platform API provides several predefined annotation types in the `java.lang` package that can readily be used in the source code. Selected predefined annotations are discussed in this section, but they should already be familiar as they have been previously used in this book. The reader is encouraged to consult the Java SE Platform API documentation for further details.

It is important to note that the standard annotations presented here are all optional, but programmers are encouraged to use them as they improve the quality of the code. If used, the compiler can aid in verifying certain aspects of the code at compile time.

The `@Override` Annotation

The marker annotation `@Override` can be used to indicate that *the annotated method in a subtype must override an inherited method from a supertype*. If that is not the case, the compiler issues an error.

Click here to view code image

```
public class A {
  public void doThings() {}
}
public class B extends A {
    @Override
    public void dothings() {}          // Wrong method name.
  }
```

Without the `@Override` annotation, the class `B` above will compile successfully even though the method `dothings()` does not really override the method `doThings()` in the superclass `A`. With the `@Override` annotation, the compiler will flag an error that the method does not fulfill the criteria for overriding methods—in this case, the method names do not match.

Basically, the `@Override` annotation when applied to a method helps to catch any errors in an attempt to override some method in its supertype at compile time, ensuring that the overriding method satisfies the criteria for method overriding (§5.1, p. 196).

A common mistake is to inadvertently overload a method from the supertype when the intention is to override it. A classic example is the `equals()` method from the `Object` class that has the header:

Click here to view code image

```
  public boolean equals(Object obj)
```

Instead, the method declaration at (1) is declared in the `Gizmo` class below, leading to subtle bugs in the code:

Click here to view code image

```
  class Gizmo {
    public boolean equals(Gizmo obj) { // (1) Overloaded, as parameter doesn’t m
      // ...
    }
  }
```

Best practices advocate that the `@Override` annotation is always used when overriding a method.

The API of the `Override` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
  @Target(ElementType=METHOD)
  @Retention(RetentionPolicy=SOURCE)
  public @interface Override
```

Note that the `Override` annotation type can only be applied to a method. Its retention policy is `SOURCE`, meaning it is discarded by the compiler once the code is validated—in other words, it is not recorded in the class file and therefore not available at runtime.

The `@FunctionalInterface` Annotation

The marker annotation `@FunctionalInterface` is designed to ensure that a given interface is indeed functional—that is, it contains exactly one abstract method that is either specified or inherited by the interface; otherwise, it will report an *error*.

Click here to view code image

```
@FunctionalInterface
public interface FIX {       // Compiles without errors.One abstract method.
  void doThings();
}

@FunctionalInterface
public interface FIXIt extends FIX {  // Compiles without errors. Abstract met
}                                     // inherited.

@FunctionalInterface
public interface FIY {       // Compile-time error! More than one abstract met
  void doThings();
  void doOtherThings();
}

@FunctionalInterface
public interface FIZ {       // Compile-time error! No abstract method.
}
```

Without the `@FunctionalInterface` annotation, the interfaces above will compile successfully as they are all valid interface declarations. However, the presence of the `@FunctionalInterface` annotation will ensure that the interface has exactly one abstract method. For more details on declaring functional interfaces, see §13.1, p. 675.

It is not mandatory that all interfaces that have exactly one abstract method should be marked with the `@FunctionalInterface` annotation. For example, the `Comparable<E>` interface qualifies as a functional interface but is not marked with the `@FunctionalInterface` annotation in the `java.lang` package API. Typically, only those interfaces whose implementation is provided by lambda expressions or method references are marked with this annotation. However, whether marked with this annotation or not, the compiler will treat interfaces with exactly one abstract method as functional interfaces.

The API of the `FunctionalInterface` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target(TYPE)
```

```
public @interface FunctionalInterface
```

Note that the annotation `@FunctionalInterface` can only be applied to a target that is a type declaration (`ElementType.TYPE`)—in particular, *only to an interface declaration*; otherwise, the compiler will flag an error. Its retention policy is `RUNTIME`, and therefore it is recorded in the class file and available at runtime. The `javadoc` tool will include the annotation in the documentation it generates, as the annotation has the meta-annotation `@Documented`.

The `@Deprecated` Annotation

The annotation `@Deprecated` is designed to discourage the use of certain declarations that were previously allowed. The deprecated code can be declarations of methods, constructors, fields, local variables, packages, modules, parameters, classes, interfaces, and enums, to which this annotation is applied.

It is typically used to indicate that certain code is considered legacy and should no longer be used for various reasons: API design changes, deficiencies in legacy implementations, availability of a better alternative approach that could be more stable, have better performance, and be less likely to cause errors or unwanted side effects. The purpose of the `@Deprecated` annotation is to caution developers that certain code is obsolete and is considered to be removed from future versions. However, the annotation does not prevent such obsolete code from being used in a program. Instead, the compiler can produce a deprecation warning when compiling code that utilizes declarations that have been marked as deprecated.

Although strictly not required, developers are strongly advised to document the reason for deprecating a program element using the Javadoc annotation `@deprecated` in a Javadoc comment. Usually such documentation should describe a reason why certain code has been deprecated and suggest an alternative replacement API, as well as point out any differences in the way this suggested API works or is supposed to be used.

There are two annotation type elements defined by the `Deprecated` annotation type:

Click here to view code image

```
String  since()      default "";
boolean forRemoval() default false;
```

The value of the `since()` element in the `Deprecated` annotation type is used to indicate the Java version when deprecation first occurred. The `boolean` value of the `for-Removal()` element, when set to `true`, indicates the intention to remove the code marked with this annotation in future versions—called *terminally deprecated*. If the value is `false`, the intent is to discourage the use of the deprecated program element, but it does not indicate an imminent intent to remove it—called *ordinarily deprecated*. As both annotation type elements are defined with default values, they can be omitted in the annotation, but it is strongly advised to provide this information for deprecated code.

Some IDEs show the name of the deprecated program element by a strikethrough, as shown in Example 25.1. The compiler can provide more details of deprecated code usage if the `-deprecation` or `-Xlint:deprecation` option is used to compile the code, as seen from the compiler output in Example 25.1. The warnings relate to the usage of the deprecated method `initAuthentication()` and the deprecated `static` field `SIGN_IN_ATTEMPTS`, respectively.

Example 25.1 *Using the* `@Deprecated` *Annotation*

Click here to view code image

```
class LoginHandler {
  /**
   * The field SIGN_IN_ATTEMPTS has now been deprecated,
   * and replaced by more appropriate name.
   * @deprecated Use ACCOUNT_LOCKOUT_THRESHOLD instead.
   */
  @Deprecated(since = "10", forRemoval = false)         // Ordinarily deprecat
  public static final int SIGN_IN_ATTEMPTS = 4;         // static field.
  public static final int ACCOUNT_LOCKOUT_THRESHOLD = 4;

  /**
   * The following method has now been deprecated.
   * @deprecated Use initTwoFactorAuthentication() instead
   */
  @Deprecated(since = "10", forRemoval = true)          // Terminally deprecat
  public void initAuthentication() { /* ...*/ }         // instance method.

  public void initTwoFactorAuthentication() { /* ... */ }
}

public class UserLogin {
  public static void main(String[] args) {
    LoginHandler lh = new LoginHandler();
    lh.initAuthentication();                            // Removal warning
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
  }
}
```

Compiling the program:

Click here to view code image

```
>javac -deprecation UserLogin.java
UserLogin.java:24: warning: [removal] initAuthentication() in LoginHandler has
been deprecated and marked for removal
    lh.initAuthentication();                            // Removal warning
      ^

UserLogin.java:25: warning: [deprecation] SIGN_IN_ATTEMPTS in LoginHandler has
been deprecated
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
                                   ^
2 warnings
```

The following example of a deprecated method can be found in the `Object` class API:

Click here to view code image

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

The example above shows that the use of the `finalize()` method is discouraged and appropriate documentation provides reasons for it, as well as suggests alternative approaches to implementing its functionality. Starting from Java 9, the compiler produces *a deprecation warning* when compiling classes that invoke or override this method.

Other examples to note are the constructors of the wrapper classes which are now deprecated (§8.3, p. 429).

The API of the `Deprecated` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Documented
@Retention(RUNTIME)
@Target({CONSTRUCTOR,FIELD,LOCAL_VARIABLE,METHOD,PACKAGE,MODULE,PARAMETER,TYPE
public @interface Deprecated
```

The primary use of the `@Deprecated` annotation is to mark code that is deprecated so that its use is discouraged. Its retention policy is `RUNTIME` so that it is retained by the compiler, making it possible to dynamically discover whether some code is deprecated through the use of the Reflection API. The contexts in which it can be used are practically all declarations.

As the declaration of the `Deprecated` annotation type specifies the meta-annotation `@Documented`, its use will also be documented by the `javadoc` tool as mentioned earlier. It is recommended to use this annotation in conjunction with the Javadoc annotation `@deprecated` in a Javadoc comment. A screenshot showing partial documentation generated for the deprecated method `initAuthentication()` of the `Login-Handler` class is shown in Figure 25.1.

Figure 25.1 *API Documentation of a Deprecated Method*

The `@SuppressWarnings` Annotation

The annotation `@SuppressWarnings` can be used to suppress different kinds of warnings from the compiler in code that would otherwise result in these warnings being issued (§11.13, p. 623).

The `SuppressWarnings` annotation type is an *array-valued single-element annotation type*—that is, it defines a `value()` element of type `String[]`. It is used to indicate which kinds of warnings should be suppressed. Typical kinds of warnings that can be specified for the `value` array are `"unchecked"` and `"deprecation"`. *Unchecked warnings* caution about mixing legacy and generic code that might result in heap pollution (§11.13, p. 630). *Deprecation warnings* result from usage of deprecated code (p. 1580). Different Java compilers may provide different kinds of warnings that can be suppressed.

The `@SuppressWarnings` annotation can be applied at different nested levels of a program construct—for example, when it is applied to a class, it will suppress warnings of a specific kind for the entire class. It is advisable to apply the `@SuppressWarnings` annotation at the deepest nested level possible, such as a specific method, or a variable that may be the cause of the compiler warnings that a programmer wishes to suppress. Specifying the same kind of warning multiple times is permissible, but only the first occurrence of the name is applied and any other occurrences of this name are ignored.

In the code below, suppressing unchecked warnings in the method is redundant, as that is already the case since its class also suppresses unchecked warnings. However, deprecation warnings are only suppressed for the annotated method in the example below. Note also that the compiler will only suppress unchecked warnings relating to the declaration of the method, but it will *not* suppress any unchecked warnings at the call sites for this method, in contrast to the `@SafeVarargs` annotation (p. 1585).

Click here to view code image

```
@SuppressWarnings("unchecked")
```

```
public class PhoneCenter {

  @SuppressWarnings({"unchecked", "deprecation"})
  public void callLandline() {}

}
```

Example 25.2 illustrates suppressing warnings with the `@SuppressWarnings` annotation. The intention is to suppress both unchecked and deprecation warnings. To show the potential problems in the code, it is first complied without the `@SuppressWarnings` annotation using appropriate compiler options. The compiler output shows two unchecked warnings and two deprecation warnings in the `undisciplined-Method()` and the overridden `finalize()` method, respectively. With these issues in the code, it is hardly recommended to suppress the warnings using the `@Suppress-Warnings` annotation; doing so will suppress them, but it will not solve the potential problems in the code.

Example 25.2 *Using the* `@SuppressWarnings` *Annotation*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

@SuppressWarnings(value={"unchecked", "deprecation"})  // (1)
public class ATSuppressWarnings {
  /** Mixing legacy code and generic code. */
  public void undisciplinedMethod() {
    List wordList1 = new ArrayList<String>();  // Assigning parameterized type
                                               // to raw type.
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
    wordList1.add("911");                      // (3) Unchecked call
    wordList2.add("119");                      // OK
  }

  /** Overriding and using a deprecated method. */
  @Override
  public void finalize() throws Throwable { // (4) Overriding a deprecated met
    super.finalize();                       // (5) Usage of deprecated method.
  }
}
```

Compiling without the `@SuppressWarnings` annotation at (1) is shown below:

Click here to view code image

```
>javac -Xlint:unchecked -deprecation ATSuppressWarnings.java
ATSuppressWarnings.java:9: warning: [unchecked] unchecked conversion
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
                             ^
  required: List<String>
  found:    List
ATSuppressWarnings.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    wordList1.add("911");                      // (3) Unchecked call
                 ^
  where E is a type-variable:
    E extends Object declared in interface List
ATSuppressWarnings.java:16: warning: [deprecation] finalize() in Object has be
deprecated
  public void finalize() throws Throwable { // (4) Overriding a deprecated met
              ^
ATSuppressWarnings.java:17: warning: [deprecation] finalize() in Object has be
deprecated
    super.finalize();                       // (5) Usage of deprecated method.
         ^
4 warnings
```

Suppressing warnings is generally considered to be potentially dangerous, as this might hide potential problems within the code. For example, suppressing deprecation warnings may result in a programmer not noticing that deprecated code is in use and may lead to the program not compiling in future versions of Java because the deprecated code has been removed. Regardless, the `@SuppressWarnings` annotation should not be used unless it has been manually verified that it is safe to do so. Any warnings not suppressed must always be looked into.

The API of the `SuppressWarnings` annotation type from the `java.lang` package is shown below:

Click here to view code image

```
@Target({TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE,MODULE})
@Retention(SOURCE)
public @interface SuppressWarnings
```

The annotation is primarily used for suppressing various warnings that the compiler would otherwise issue. Note that the annotation `@SuppressWarnings` can be applied to declarations of class, interface, enum type, fields, methods, parameters, constructors, local variables, and modules. Its retention policy is `SOURCE` and thus it is discarded by the compiler once the code is validated—in other words, it is not recorded in the class file and therefore not available at runtime.

The `@SafeVarargs` Annotation

The marker annotation `@SafeVarargs` is used to instruct the compiler to suppress unchecked warnings that would otherwise be issued if the declaration or the invocation of a variable arity method or constructor has a variable arity parameter of a non-reifiable element type, as this can lead to heap pollution (§11.13, p. 630).

It is entirely the responsibility of the programmer to ensure that the variable arity method or constructor is well formed to prevent *heap pollution* before using the `@SafeVarargs` annotation to suppress the warnings. Note that the `@SafeVarargs` annotation suppresses any unchecked warnings both for the declaration and all call sites of the variable arity method or constructor.

In order to prevent heap pollution occurring in a subclass, a variable arity method cannot be overridden. The compiler will only allow this annotation on a method that is either `static`, `private`, or `final`. Constructors are not a problem in this regard as they cannot be overridden.

Example 25.3 below illustrates using the `@SafeVarargs` annotation. It is compiled with and without the `@SafeVarargs` annotation at (1). The output without the @`Safe-Varargs` annotation shows that two warnings are generated: the first one where the method `printList()` is declared at (2) and the second one at the call site for this method at (3). Since the variable arity method `printList()` is well formed, there is no risk in using the annotation.

Example 25.3 *Using the* `@SafeVarargs` *Annotation*

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
public class SafeVarargsTest{
  @SafeVarargs                                              // (1)
  private static void printList(List<String>... toys) { // (2) List<String>[]
    for (List<String> toy : toys) {
      System.out.println(toy);
    }
  }

  public static void main(String[] args) {
    List<String> tList = new ArrayList<String>();
    tList.add("vaporizer"); tList.add("slime gun");
    printList(tList);                                       // (3) new List<String
  }
}
```

Without `@SafeVarargs` at (1):

Click here to view code image

```
>javac SafeVarargsTest.java
```

```
Note: SafeVarargsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

>javac -Xlint:unchecked SafeVarargsTest.java
SafeVarargsTest.java:5: warning: [unchecked] Possible heap pollution from para
terized vararg type List<String>
  private static void printList(List<String>... toys) { // (2) List<String>[]
                                                 ^

SafeVarargsTest.java:14: warning: [unchecked] unchecked generic array creation
varargs parameter of type List<String>[]
    printList(tList);                                    // (3) new List<String
             ^

2 warnings
```

With `@SafeVarargs`:

Click here to view code image

```
>javac SafeVarargsTest.java
>java SafeVarargsTest
[vaporizer, slime gun]
```

The API of the `SafeVarargs` annotation type from the `java.lang` package is shown below:

```
@Documented
@Retention(RUNTIME)
@Target({CONSTRUCTOR,METHOD})
public @interface SafeVarargs
```

The primary use of the `@SafeVarargs` marker annotation is to suppress unchecked warnings alerting to potential heap pollution that can result from a variable arity parameter of a non-reifiable element type in a variable arity method or constructor.

The `@SafeVarargs` marker annotation has the `RUNTIME` retention policy and is thus not discarded by the compiler, making it possible to dynamically discover it through the use of the Reflection API. It can be used to annotate the declaration of a constructor or a method—the compiler ensures that the method is either `static`, `private`, or `final`. Its use will also be documented by the `javadoc` tool.

Table 25.3 provides a summary of selected standard annotations.

Table 25.3 *Summary of Selected Standard Annotations*

[illegible] es of target				
[illegible]	[illegible]	[illegible]	[illegible]	
[illegible] not have exactly one abstract method				
`CONSTRUCTOR,`				

Warnings when deprecated code is used or overridden in nondeprecated code

`String since() default ""; boolean forRemoval() default false;`

`CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE`

Warnings are suppressed

`String[] value();`

`CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, MODULE, PARAMETER, TYPE`

Marker annotation

`CONSTRUCTOR, METHOD` (*which must be* `static`, `private`, *or* `final`)

Error when varargs parameter is specified or when applied to a method that is not either `static`, `private`, or `final`; otherwise warnings are suppressed

Other Annotations

Many other annotations used in various contexts are defined in the language. Additional annotations may be provided with various extended APIs that are not part of the core Java language, but nevertheless are often used in practical programming, especially in server-side Java applications deployed in the Java Enterprise Edition or Micro Profile environments. Examples include the Java Persistence (JPA), Container Dependency Injection (CDI), and Bean Validation APIs. These APIs go beyond the Java SE Edition and are therefore not on the Java SE certification exam. However, they are essential for developing real-world Java applications.

25.6 Processing Annotations

It is possible to discover annotations on program elements at runtime using the Reflection API found in the `java.lang.reflect` package. As this topic is beyond the scope of this book, we provide a brief introduction to the Reflection API—in particular, how to discover annotations on a class and its members.

Figure 25.2 shows a few selected classes and interfaces from the Reflection API. At runtime, an instance of the class `Class<T>` represents the type `T` in a Java program. Classes, interfaces, enums, annotations, and primitive types are all represented by `Class` objects at runtime. For example, the class `NuclearPlant` is represented by a `Class` object which can be referenced by `NuclearPlant.class` and whose type is `Class<NuclearPlant>` at compile time to ensure type-safety before type erasure removes the type parameter.

Figure 25.2 *Selected Types from the Reflection API*

A `Class<T>` object representing a class can be queried for members that are declared in the class:

Click here to view code image

```
Constructor<?>[] cons = classobj.getDeclaredConstructors();
Method[] methods      = classobj.getDeclaredMethods();
Field[] fields        = classobj.getDeclaredFields();
```

The classes `Class<T>`, `Method`, `Field`, and `Constructor<T>` implement the `java.lang.reflect.AnnotatedElement` interface, as shown in Figure 25.2. The method `getDeclaredAnnotations()` of the `AnnotatedElement` interface can be used to obtain the annotations applied to an annotated element. All annotation types implicitly implement the `java.lang.annotation.Annotation` interface. The method `getDeclared-Annotations()` of the `Annotation` interface returns an array of `Annotation` containing the annotations applied to an annotated element. Using this method, we can obtain the annotations on a class or any member declared in the class.

Click here to view code image

```
Annotation[] annotations = annotatedElement.getDeclaredAnnotations();
```

If we are interested in a particular annotation on an annotated element, we can use the `getDeclaredAnnotationsByType()` method of the `AnnotatedElement` interface. The code below returns all `@TaskInfo` annotations applied on the `NuclearPlant` class. Since the `@TaskInfo` annotation is not repeatable, the array returned will have at most one `TaskInfo` object, which can be queried by calling the methods declared in the `TaskInfo` type declaration.

Click here to view code image

```
TaskInfo[] tias = NuclearPlant.class.getDeclaredAnnotationsByType(TaskInfo.cla
```

Of course, in order for an annotation to be discoverable at runtime, its retention policy must be `RUNTIME`.

In Example 25.4, the class `NuclearPlant` and its members are marked with annotations which have previously been declared in this chapter: `@Pending` and `@TaskInfo`. In addition, the standard annotations `@Deprecated` and `@Override` are also applied to some methods.

Example 25.4 *Processing Annotations*

Click here to view code image

```
@Pending
@TaskInfo(
    priority   = TaskInfo.TaskPriority.HIGH,
    taskDesc   = "Class for running a nuclear reactor.",
    assignedTo = {"Tom", "Dick", "Harriet"}
    )
public class NuclearPlant {

  @Pending
  public NuclearPlant() {}

  @Deprecated(forRemoval = true, since = "8")
  public boolean outOfProduction;

  @Deprecated(since = "10")
  public void notInUse() {}

  @Pending
  @TaskInfo(
      taskDesc   = "Procedure for nuclear reactor shutdown",
      assignedTo = {"Tom", "Harriet"}
      )
  public void shutDownNuclearReactor() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Exchange nuclear rods",
      assignedTo = {"Tom", "Dick"}
      )
  public void changeNuclearRods() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Adjust nuclear fuel",
      assignedTo = {"Harriet"}
      )
  public void adjustNuclearFuel() {}

  @TaskInfo(
      taskDesc   = "Start nuclear reactor",
      assignedTo = "Dick"
      )
  public void startNuclearReactor() {}

  @Pending
  @Override
  public String toString() {
    return "TBD";
  }
}
```

Click here to view code image

```
import static java.lang.System.out;

import java.lang.reflect.AnnotatedElement;
import java.util.Arrays;
import java.util.stream.Stream;

public class TaskInfoAnnotationProcessor {
  public static void printTaskInfoAnnotation(AnnotatedElement... elements) {//
    Stream.of(elements)                                                     //
      .filter(ae -> ae.isAnnotationPresent(TaskInfo.class))                 //
      .peek(ae -> out.printf("%s annotation for '%s':%n",                   //
          TaskInfo.class.getName(), ae))
      .flatMap(ae -> Stream.of(
                       ae.getDeclaredAnnotationsByType(TaskInfo.class)))    //
      .forEach(a -> {                                                       //
        out.printf("  Task description: %s%n", a.taskDesc());
        out.printf("  Priority: %s%n", a.priority());
        out.printf("  Assigned to: %s%n", Arrays.toString(a.assignedTo()));
      });
  }

  public static void main(String[] args) {
    Class<?> classobj = NuclearPlant.class;                                 //
    printTaskInfoAnnotation(classobj);                                      //
    printTaskInfoAnnotation(classobj.getDeclaredMethods());                 //
  }
}
```

Output from the program:

Click here to view code image

```
TaskInfo annotation for 'class NuclearPlant':
  Task description: Class for running a nuclear reactor.
  Priority: HIGH
  Assigned to: [Tom, Dick, Harriet]
TaskInfo annotation for 'public void NuclearPlant.shutDownNuclearReactor()':
  Task description: Procedure for nuclear reactor shutdown
  Priority: NORMAL
  Assigned to: [Tom, Harriet]
TaskInfo annotation for 'public void NuclearPlant.changeNuclearRods()':
  Task description: Exchange nuclear rods
  Priority: LOW
  Assigned to: [Tom, Dick]
TaskInfo annotation for 'public void NuclearPlant.adjustNuclearFuel()':
  Task description: Adjust nuclear fuel
  Priority: LOW
  Assigned to: [Harriet]
TaskInfo annotation for 'public void NuclearPlant.startNuclearReactor()':
  Task description: Start nuclear reactor
  Priority: NORMAL
  Assigned to: [Dick]
```

In Example 25.4, the method `printTaskInfoAnnotation()` at (1) in the `TaskInfoAnnotationProcessor` class can be used to specifically discover `TaskInfo` annotations on program elements. Its parameter is an array of `AnnotatedElement`—that is, it contains program elements that are presumably marked with annotations. A stream of `AnnotatedElement` to process each annotated element is created at (2) from the array of `AnnotatedElement` that is passed as an argument to the method. This stream is filtered at (3) to discard any annotated element that is not marked with the `TaskInfo` annotation. A header is printed at (4) to indicate which annotated element is being processed for `TaskInfo` annotation. At (5) the method `getDeclaredAnnotationsByType()` returns an array of

`TaskInfo`, containing any `TaskInfo` annotations applied to the current annotated element. This array of `TaskInfo` is flattened to create a stream of `TaskInfo` annotations. The element values of a `TaskInfo` annotation are extracted at (6) by calling the methods specified in the annotation type elements declared in the `TaskInfo` annotation type.

The method `printTaskInfoAnnotation()` is called at (8) and (9) to process `TaskInfo` annotations for the class `NuclearPlant` and its methods, respectively. The program output shows the result of running this annotation processor on the `NuclearPlant` class.

In Example 25.5, the class `AnnotationPrinter` implements a more general annotation processor that prints all annotations applied on a class and its members. The numbers below refer to the numbered lines in the code in Example 25.5.

(1) The `printAllAnnotations()` method is passed the `Class` object of the class whose annotations should be printed.

(2) through (5): The method `printAnnotatedElements()` is successively called to print any annotations on the class, its constructors, its methods, and its fields.

(6) The method `printAnnotatedElements()` accepts elements that implement the `AnnotatedElement` interface. Recall that this is the case for the `Class<T>`, `Constructor<T>`, `Method`, and `Field` classes (Figure 25.2).

(7) A stream of `AnnotatedElement` is created.

(8) The name of the `AnnotatedElement` is printed.

(9) The method `getDeclaredAnnotations()` returns an array with the annotations applied to an `AnnotatedElement`, which is flattened into a stream of `Annotation`.

(10) The text representation of an `Annotation` is printed.

The program output shows the results of running the annotation processor on the `NuclearPlant` class. It is worth running this annotation processor on classes that we have seen in this chapter, keeping in mind that any annotation applied in the code must have `RetentionPolicy.RUNTIME` in order for it to be processed at runtime. For instance, the `@Override` annotation on the `toString()` method of the `NuclearPlant` class is not in the program output, as it has `RetentionPolicy.SOURCE`.

Example 25.5 *Annotation Processor*

Click here to view code image

```
import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
import java.util.stream.Stream;

public class AnnotationPrinter {

  public static void printAllAnnotations(Class<?> classobj) {              //
    printAnnotatedElements(classobj);                                      //
    printAnnotatedElements(classobj.getDeclaredConstructors());            //
    printAnnotatedElements(classobj.getDeclaredMethods());                 //
    printAnnotatedElements(classobj.getDeclaredFields());                  //
  }

  public static void printAnnotatedElements(AnnotatedElement... elements) {//
    Stream.of(elements)                                                    //
      .peek(ae -> System.out.printf("Annotations for \'%s\':%n", ae))      //
      .flatMap(ae -> Stream.of(ae.getDeclaredAnnotations()))               //
      .forEach(a -> System.out.println("  " + a));                         //
  }
}
```

Click here to view code image

```
public class AnnotationClient {
  public static void main(String[] args) {
    AnnotationPrinter.printAllAnnotations(NuclearPlant.class);
  }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Annotations for 'class NuclearPlant':
  @Pending()
  @TaskInfo(priority=HIGH, taskDesc="Class for running a nuclear reactor.",
            assignedTo={"Tom", "Dick", "Harriet"})
Annotations for 'public NuclearPlant()':
  @Pending()
Annotations for 'public void NuclearPlant.shutDownNuclearReactor()':
  @Pending()
  @TaskInfo(priority=NORMAL, taskDesc="Procedure for nuclear reactor shutdown"
            assignedTo={"Tom", "Harriet"})
Annotations for 'public void NuclearPlant.notInUse()':
  @java.lang.Deprecated(forRemoval=false, since="10")
Annotations for 'public void NuclearPlant.changeNuclearRods()':
  @TaskInfo(priority=LOW, taskDesc="Exchange nuclear rods",
            assignedTo={"Tom", "Dick"})
Annotations for 'public void NuclearPlant.adjustNuclearFuel()':
  @TaskInfo(priority=LOW, taskDesc="Adjust nuclear fuel", assignedTo={"Harriet
Annotations for 'public void NuclearPlant.startNuclearReactor()':
  @TaskInfo(priority=NORMAL, taskDesc="Start nuclear reactor",
            assignedTo={"Dick"})
Annotations for 'public java.lang.String NuclearPlant.toString()':
  @Pending()
Annotations for 'public boolean NuclearPlant.outOfProduction':
  @java.lang.Deprecated(forRemoval=true, since="8")
```

Review Questions

25.1 Which of the following statements are true about annotations? Select the two correct answers.

a. Annotations are compiled as classes.

b. Annotations are used to provide metadata for Java program elements.

c. An annotation cannot be applied to other annotations.

d. An annotation cannot be an element type in another annotation.

25.2 Which of the following statements is true about the value of `ElementType` that can be specified for the meta-annotation `@Target` in an annotation type declaration— that is, the target of an annotation type?

Select the one correct answer.

a. `ElementType.TYPE` means the annotation type cannot be applied to annotations.

b. `ElementType.TYPE_PARAMETER` means the annotation type cannot be applied to generic types.

c. `ElementType.FIELD` means the annotation type can be applied to constants.

d. `ElementType.METHOD` means the annotation type can be applied to constructors.

25.3 Given the following code:

```
@Test3Annotation              // (1)
public class Test3 { }
```

Which of the following annotation types are compatible with the annotation applied at (1)?
Select the two correct answers.

a. Click here to view code image

```
public @interface Test3Annotation { }
```

b. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target(PACKAGE)
public @interface Test3Annotation { }
```

c. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({FIELD,TYPE})
public @interface Test3Annotation { }
```

d. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({PACKAGE,TYPE_USE}) public
@interface Test3Annotation { }
```

e. None of the above

25.4 Given the following code:

Click here to view code image

```
@interface Test4Annotation {
  int[] value() default {1,2,3};
}
```

and

Click here to view code image

```
// (1) INSERT ANNOTATION HERE
public class Test4 { }
```

Which of the following are incorrect ways to apply `@Test4Annotation` at (1)? Select the two correct answers.

a. `@Test4Annotation(value=7)`
b. `@Test4Annotation(value=7,8,9)`
c. `@Test4Annotation({7,8,9})`
d. `@Test4Annotation`
e. `@Test4Annotation()`
f. `@Test4Annotation(8)`
g. `@Test4Annotation(8,9)`

25.5 Given the following code:

Click here to view code image

```
@Test5Annotation
public class Test5<@Test5Annotation(value="a") T> {
  @Test5Annotation({"a","b"})
  private T var;
  public Test5(T var) {
  this.var = var;
  }
}
```

Which of the following annotation types is compatible with the annotation `@Test5Annotation`?
Select the one correct answer.

a. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE})
@interface Test5Annotation {
  String[] value() default {"x"};
}
```

b. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
  String[] value() default "x";
}
```

c. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
  String[] values() default {"x"};
}
```

d. Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, TYPE_PARAMETER})
@interface Test5Annotation {
  String[] values() default "x";
}
```

25.6 Which of the following statements is true about annotations being reflected in the class documentation generated by the `javadoc` tool?

Select the one correct answer.

a. Annotations applied at the class level are reflected in the class documentation.

b. Annotations applied at the method level are reflected in the class documentation.

c. Annotations applied to public methods are reflected in the class documentation.

d. Annotations are not reflected in the class documentation by default.

25.7 Which of the following annotation types are correctly declared? Select the two correct answers.

a. Click here to view code image

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

b. Click here to view code image

```
public @interface Location {
  String value() default "London";
  int[] coordinates() default 1;
}
```

c. Click here to view code image

```
public @interface Location {
  String value();
  int[] coordinates default {1,1};
}
```

d. Click here to view code image

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

e. Click here to view code image

```
public @interface Location {
  String value();
  int[] coordinates() default {1,1};
}
```

25.8 Given the following code:

Click here to view code image

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({FIELD, CONSTRUCTOR, TYPE})
public @interface Descriptor {
  String value();
  String[] details() default "";
}
```

Which of the following applications of this annotation type are valid? Select the three correct answers.

a. Click here to view code image

```
@Descriptor("Music to play")
public interface Playable { }
```

b. Click here to view code image

```
public enum Style {
  @Descriptor(value="Rock music", details={"listen"})
  ROCK,
  @Descriptor(value="POP music", details="dance")
  POP,
  @Descriptor(value="Jazz music", details={"listen", "dance"})
  JAZZ,
  @Descriptor("Classic music")
  CLASSIC;
}
```

c. Click here to view code image

```
public class Music {
   @Descriptor(value="Music to play")
   Music m = new Music();
}
```

d. Click here to view code image

```
@Descriptor("Music to play",details={"listen","dance"})
public class Music {
  Music m = new Music();
}
```

e. Click here to view code image

```
public class Player {
  @Descriptor("Music to play")
  public static void main(String[] args) {
    Music m = new Music();
  }
}
```

f. Click here to view code image

```
public class Player {
  public static void main(String[] args) {
     @Descriptor(value="Music to play",details={"listen","dance
     new Music();
  }
}
```

25.9 Given the following code:

Click here to view code image

```
import java.lang.annotation.*;
@Repeatable(Container.class)
public @interface Containee { int value(); }
```

and

Click here to view code image

```
public @interface Container {
  // (1) INSERT CODE HERE
}
```

Which of the following code fragments inserted at (1) will allow the annotation types to compile?
Select the one correct answer.

a.

```
String name();
  Containee[] value();
```

b.

```
String name() default "x";
    Containee[] value();
```

c.

```
String name() default "x";
    Containee[] values();
```

d. Click here to view code image

```
String name() default "x";
    Containee value() default @Containee;
```

e. Click here to view code image

```
String name() default "x";
    Containee[] values() default {@Containee(1), @Containee(2)}
```

f. Click here to view code image

```
String name();
    Containee[] values() default {@Containee(1), @Containee(2)}
```

25.10 Given the following code:

Click here to view code image

```
@Folder("/data")
@Folder(value="/tmp", temp=true)
public class Storage { }
```

Which of the following declarations of the `Folder` and `Folders` annotation types is valid?
Select the one correct answer.

a. Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

and

Click here to view code image

```
public @interface Folders {
  Folder[] paths();
}
```

b. Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp();
}
```

and

Click here to view code image

```
public @interface Folders {
  Folder[] value();
}
```

c. Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

and

```
public @interface Folders {
  Folder[] value();
}
```

d. Click here to view code image

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String path();
  boolean temp() default true;
}
```

and

```
public @interface Folders {
  Folder[] value();
}
```

Secure Coding 26

Chapter Topics

- An overview of application security
- Identifying security threat categories
- Understanding denial-of-service (DoS) attacks and their mitigation
- Recognizing sensitive information leakages, and applying value obfuscation and data encryption
- Understanding code injection and the importance of input validation
- Understanding code corruption and its mitigation through practices of encapsulation, data integrity, robustness, judicious extensibility, and immutability
- Understanding the basics of Java security policies and how to execute privileged code
- Presenting additional security guidelines on accessing the file system, utilizing deserialization properly, and the importance of good class design practices

Java SE 11 Developer Exam O	
[26.1] Develop code that mitigates security threats such as denial of service, code injection, input validation and ensure data integrity	*§26.1, p. 1600* *§26.2, p. 1602*
[26.2] Secure resource access including filesystems, manage policies and execute privileged code	*§26.3, p. 1608* *§26.4, p. 1610*

The topic of secure coding is marginal compared to the core topics of the Java SE 11 Developer exam. No attempt is made here to cover this important topic comprehensively in a single chapter. Selected topics in secure coding are covered to the extent of their relevance for Java SE 11 certification. We encourage the reader to consult literature on *software security* that encompasses essential security topics in developing secure and reliable systems, not just secure coding.

26.1 Application Security Overview

This chapter is quite different from other chapters in this book because of the nature of the topic that is covered here. So far this book has covered Java language constructs and APIs that programmers use to implement the business logic of the application. Application designers and architects describe business logic requirements as *functional requirements* (FRs). Such requirements must be implemented to satisfy the business reasons for this application and implement its functionality— in other words, they answer the question of *what* this application should do. However, there is more to application design than that. There is another set of requirements that application designers and architects should consider, known as *nonfunctional requirements* (NFRs). These include things that are not directly related to the question of *what* the application is supposed to do, but rather characterize *how* it is supposed to work—for example, application performance, scalability, user interface ergonomics, maintainability, and security.

Securing and protecting applications is an important NFR and is usually treated as a separate set of concerns in the overall development cycle. NFR-related design decisions are a complex set of considerations that have to be resolved by a fairly large group of participants, including system and application architects, application designers, and even business stakeholders, not just programmers. Programmers usually are responsible for implementing design and architecture decisions that are made for them. However, it is still quite beneficial for a programmer to be able to understand and appreciate the logic behind the NFRs. Therefore, a reasonable level of understanding of the security principles of application design is advised for any practical programmer. Such an appreciation may help programmers reduce cases when code they have written has to be adjusted or reengineered to satisfy NFRs, including security design decisions.

It is important to understand that security enforcement always comes at a price. This could be a direct financial cost of actually investing in designing, coding, and maintaining a secure environment, but also other overheads, such as decreased performance or decreased throughput of the application that could be caused by data encryption and decryption, or verification and validation of values, and so on.

Improving application security can have implications on its usability—for example, making it less convenient for users to access the application or use some of its functionality because security measures may require users to overcome extra hurdles, such as using two-factor authentication or connecting using virtual private network (VPN). Therefore, a cost-benefit analysis should always be performed as part of the application security design process.

The question that needs to be addressed is this: Is extra protection worth the increase in the up-front development investment, potential user inconvenience, increased maintenance costs, and possible adverse impact on performance and scalability? This is not an easy question to answer because it may involve considerations that go far beyond software design concerns. For example, data leaks caused by security breaches can carry financial and reputation risks for a company, or can even put the company in breach of regulatory requirements and thus have legal implications. Overall security design is often a compromise that is a reflection of a precarious and delicate balance between improved protection and the overhead that it entails.

Lastly, it should be noted that the overall security of the application can only be as good as its weakest link. There is no point in overinvesting in addressing a specific security concern if other related areas are not secured at a comparable level. Potential attackers will always look for the weakest link in the overall security model of the application. Don’t forget that it is not just software, but also humans that could be the weak link in security. For example, people who are not careful about keeping their password secret could pose a significant security risk depending on their role in the organization, or what information they can access.

Many of these considerations go far beyond the scope of this book because they are considered to be a responsibility of designers and architects rather than programmers. Therefore, this chapter takes a much narrower approach and only discusses issues that have a direct impact on programmers’ activities related to security implementation within an application.

It is also worth mentioning that many security concerns raised in this chapter are applicable to much more complex Java applications hosted in the Java Enterprise Edition (EE)/Jakarta and MicroProfile server environments. Thus many implementation details would fall outside the Java SE scope. However, a Java programmer should still have some degree of awareness of these issues and at least the basics techniques for how to remedy them because Java classes are portable between runtime environments, and thus some of the code written in the context of the Java SE application may eventually end up being used in the Java server environment. This chapter also avoids going into detail about environment management tasks, such as creating and maintaining SSL keys and managing keystore files, and instead keeps the reader focused on programmer-specific tasks related to security implementation. These extra topics may be of interest to developers who not only write code, but also are responsible for maintaining environments were the code will be executed. However, these topics are beyond the scope of Java SE certification.

26.2 Security Threat Categories

Security threats can be grouped into categories based on the nature of the threat and its impact on a system. Each category has its own characteristics that programmers need to be aware of, in order to address these security threats throughout the software lifecycle.

Denial-of-Service (DoS) Attacks

Denial-of-Service (DoS) attacks are a group of threats that are related to the way in which an application manages its resources—specifically, various attempts to exploit the lack of checks and restrictions around resource utilization. For example, a resource could be an open port, or a file, or memory allocation—in other words, something that an application needs for its operational requirements. The idea behind the DoS attack is to make requests to the application for its resources so that their availability for legitimate use is blocked. In DoS attack scenarios, the attacker is trying to exploit the lack of controls and restrictions that an application applies when it is allocating or accessing its resources.

The following scenarios are examples of DoS attacks:

- An attacker providing a very large file or document containing recursive references, causing the application to waste resources trying to parse such a document. This could cause the program to run out of memory, go into an infinite loop trying to handle the recursive execution, or simply slow down as it attempts to read a large document. In any case, this would impede this application's ability to process legitimate data from other sources.

- An attacker opening a connection to a port through which the application performs network interactions. Once this connection is established, the attacker can start sending or receiving information in the slowest possible manner, causing the application to block for read or write operations to complete. The attacker can also spawn a large number of such connections, with the eventual goal of causing this application to run out of its capacity to handle simultaneous connections and denying service from legitimate users.

Countermeasures that can remedy DoS attacks include the following:

- Checking file sizes before starting parsing
- Detecting recursive references and stopping data processing if such recursions occur
- Discarding suspicious documents
- Detecting how many connections a given client has opened concurrently
- Detecting and terminating excessively slow uploads and downloads
- Dropping suspicious connections
- Limiting access to program logic to only authenticated and authorized code

Sometimes it might be the case that a legitimate user struggles with a network connection due to poor network quality. This means that slow connections must still be handled by the program. Consider using asynchronous I/O capabilities to handle these types of slow connections. Asynchronous I/O is supported in Java Enterprise Edition (EE)/Jakarta and MicroProfile servers.

Sensitive Information Leakage

Sensitive information leakage is a group of threats that are related to the way in which an application maintains and transmits its data. This could be internal information storage as well as data exchanges and transmission of values across the network. Reasons behind this type of security vulnerability typically are the lack of value obfuscation, encryption, and information reduction.

For example, the application may have to handle sensitive values, such as personal information or financial details. Such values should be stored in a secure manner, and should not be transmitted through the network in clear text, or maybe not transmitted at all. Objects containing sensitive information should be expunged from memory as soon as possible. Sensitive information should not be written into logs, which includes omitting such information from exception logging. And of course, it is not recommended to serialize sensitive data. Consider performing selective serialization (§20.5, p. 1261), marking such variables as `transient`, or obfuscating values before serializing.

Nevertheless, sometimes programs do need to write sensitive data, or transmit it through the network. If that is the case, there are several techniques that should be considered to secure such data.

Value Obfuscation

Value obfuscation is a process of scrambling data, producing a digest value that can be used in place of the original data. The purpose of the digest is to act as a value substitution which is not supposed to be unscrambled.

Click here to view code image

```
try {
  String creditCard = "12345678890";
  MessageDigest md = MessageDigest.getInstance("SHA-256");  // (1)
  byte[] digest = md.digest(creditCard.getBytes());         // (2)
  String hash = (new BigInteger(1, digest)).toString(16);   // (3)
} catch (NoSuchAlgorithmException e) {
  e.printStackTrace();
}
```

This example assumes that a credit card number needs to be obfuscated. The numbered comments below correspond to the numbered lines in the code:

1. A `java.security.MessageDigest` object is created using the *SHA-256 digest algorithm*. The SHA algorithm is designed to take a variable-length input and produce a fixed-length digest result. In the case of the SHA-256 algorithm, the number 256

refers to the length of the result in bytes.

2. A digest value in the form of a `byte` array is obtained from the original value.
3. The digest value is represented as a string of fixed length.

Now this digest can be used in place of the original credit card number. It can be stored, or transmitted in the network.

Data Encryption

Another approach is to secure data using encryption. In order to perform encryption and decryption of information, a cipher algorithm and a key have to be selected. This key is then used to perform actual encryption and decryption actions, utilizing an appropriate cipher algorithm. The exact list of supported digest and encryption algorithms varies among different versions and implementations of Java, so developers should consult release-specific Java documentation for a full list of supported algorithms.

Click here to view code image

```
try {
  String creditCard = "12345678890";
  SecretKey key = KeyGenerator.getInstance("AES").generateKey(); // (1)
  Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");       // (2)
  cipher.init(Cipher.ENCRYPT_MODE, key);                         // (3)
  byte[] encryptedValue = cipher.doFinal(creditCard.getBytes()); // (4)
  GCMParameterSpec ps = cipher.getParameters()
                              .getParameterSpec(GCMParameterSpec.class);
  cipher.init(Cipher.DECRYPT_MODE, key, ps);                     // (5)
  byte[] decryptedValue = cipher.doFinal(encryptedValue);        // (6)
} catch (GeneralSecurityException e) {
  e.printStackTrace();
}
```

The numbered comments below correspond to the numbered lines in the code:

1. The key in this example is generated on the fly using the AES algorithm. However, often keys are created in advance and stored in secure SSL wallet files that implement password-protected key storage.
2. A new `javax.crypto.Cipher` object is created that is associated with a cipher algorithm.
3. A cipher is now associated with a key and initialized to be used for encryption.
4. A value is encrypted using the previously initialized cypher. There are several methods provided by the `Cipher` class that

can perform the encryption or decryption, such as `update()` and `doFinal()`. In simple cases, a single `doFinal()` method call can be used to perform encryption or decryption actions. In more complex scenarios, such as the need to handle large messages, multiple-part encryption or decryption mechanics can be used with intermediate actions performed by the `update()` method, before the invocation of the `doFinal()` method.

5. Next, a cipher parameter specification object is acquired, which is used to set up the cipher for the decryption mode. Notice that the same key that was used for encrypting the values is now used to decrypt them.
6. Finally, the cipher is used to decrypt the values.

Notice that the cipher encrypts and decrypts values as `byte` arrays, so any other type values should first be converted into a `byte` array before encryption can happen. After decryption completes, the `byte` array containing the decrypted value can be converted to the required type.

Finally, when exchanging information among different systems, consider only transmitting the minimal amount of data outside an application, if possible.

Code Injection and Input Validation

Code injection is a group of threats that are related to malicious code being injected into the application, instead of legitimate values that the application is supposed to be processing. Code injection is often related to interpreted programming languages, and situations when dynamically supplied code can be executed by the application. Usually, neither of these cases is applicable to a Java application because Java code is fully verified and compiled before execution. However, Java programs can pass executable code disguised as legitimate values that are the source of the code injection. Once again, there is a possibility that malicious executable code can be supplied instead of a regular data value that the application is supposed to handle, causing the application behavior to change when the malicious value is interpreted. Consider validating input values by checking value patterns to ensure that data appears legitimate. Also, consider sanitizing input values by modifying values in a way that prevents their interpretation as executable code.

SQL Injection

Consider the code below, which can be exploited for SQL injection (§24.4, p. 1525). It uses a basic JDBC `Statement` object at (1) to execute a query, but performs a simple concatenation to create the query string passed to the method.

Click here to view code image

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement s = connection.createStatement()) {
  String value = "whatever";
  ResultSet rs = s.executeQuery("select * from songs where name like '"
                                + value + "'");                // (1)
} catch (SQLException e) {
  e.printStackTrace();
}
```

The expectation is that this code snippet performs a query. However, this code can be exploited if the user submits a SQL statement as a parameter value, such as `"Beatles'; drop table songs;"`. To avoid this problem, consider using a `PreparedStatement` object (§24.4, p. 1526), which does not concatenate parameters, or using an `enquoteLiteral(value)` method provided by the `Statement` object to sanitize the parameter value.

Code Corruption

Code corruption is a group of threats that are related to the way in which Java code can be exploited by application developers. There is a possibility that the application logic can be broken because of a lack of encapsulation, or immutability, or loss of semantic cohesion when extending classes. Arguably, these types of issues could be considered general programming malpractice problems rather than security threats per se.

Encapsulation

A field that is declared as `public` allows developers to write code that reads and writes its value directly, which may result in data corruption and loss of integrity, even though relevant safe operations could be provided by the application. The issue is that a developer is not forced to invoke such operations if direct access to data is allowed due to lack of encapsulation.

Click here to view code image

```
public class SyncCounter {
  public int value;                                          // (1)
  public synchronized void increment()  { value++; }
  public synchronized int  getValue()   { return value; }
}
```

In the preceding code, the variable value at (1) should be `private`; otherwise, no enforcement of memory integrity would be possible. This principle is applicable to any operation—not just those that enforce synchronization (§22.4, p. 1387), but also those that validate values, or transform values, or perform any additional actions besides just setting or getting values.

Data Integrity

Restricting access to data via methods provides additional opportunities to improve overall program integrity. Operations can be used to validate values, and guard against erroneous values, such as number overflow (§2.8, p. 59). Consider using methods that guard against such overflows. It may be better for a program to throw an `ArithmeticException`, rather than silently overflow a numeric value.

Click here to view code image

```
int x = Integer.MAX_VALUE;
int y = x + 1;                              // (1)
int z = Math.addExact(x, 1);                // (2)
```

This example attempts to add 1 to the maximum `int` value. The code at (1) causes a value to overflow and wrap around (§2.8, p. 59), and the code at (2) throws an `ArithmeticException` when overflow occurs.

Similarly, a floating-point operation may result in positive or negative infinity, or result in *Not-a-number* (NaN) when attempting floating-point division by zero. Developers are advised to guard against such erroneous cases using simple `boolean` verification of operation results.

Click here to view code image

```
boolean veryLargeNum = Double.isInfinite(1/Double.MIN_VALUE); // (1)
boolean nan = Double.isNaN(0.1/0);                            // (2)
```

The code at (1) checks if the operation result is an infinitely large number, and the code at (2) checks if the operation result is NaN.

Robustness

Finally, it is worth considering guarding against references with a `null` value using an `Optional` object (§16.6, p. 940).

Click here to view code image

```
Optional<String> o = computeOptionalResult(); // (1)
String s = null;
if (o.isPresent()) {                          // (2)
  s = o.get();
}
s = o.orElse("Default");                      // (3)
s = o.orElseThrow();                          // (4)
```

The method at (1) returns an `Optional` object that can encapsulates an actual value. The caller is forced to check what value is in the `Optional` object and take appropriate action. Here are some example of such actions:

- Check whether the value is present before attempting to retrieve it from the `Optional` object.
- Substitute a default value if a value is not present in the `Optional` object.
- Throw a `NoSuchElementException` if the value is not present in the `Optional` object.

The common requirement for all of these cases to work is proper encapsulation to ensure that relevant validations and value corrections are actually applied by appropriate methods.

Extensibility

An example of a loss of semantic cohesion when extending classes is when a developer extends a class and overrides a method in such a way that the overriding method breaks semantic assumptions imposed by the supertype method (§5.1, p. 196). This may lead to erroneous interpretation of the behavior when using polymorphism. Other developers may expect the invoked method to adhere to certain semantics, but instead the invoked method would exhibit unexpected and possibly erroneous behavior.

Immutability

In order to avoid code corruption issues, consider using immutable design, declaring all variables that do not change value as `final`

(§5.5, p. 225), as well as declaring classes and methods as `final` to prevent them from being overridden in all cases where extensibility is not required, or if certain method semantics are critical for application functionality.

Click here to view code image

```
public class Authenticator {
  public boolean authenticate() {            // (1)
    /* Perform authentication logic. */
  }
}
```

The method or even the entire class above could be made `final` to ensure that no other class can extend this class or override its methods. This would prevent overriding method implementations that can disable or compromise the actual authentication process. This principle is also applicable in many other cases, such as operations that validate values, ensure memory integrity, and so on.

26.3 Java Security Policies

Security policies are designed to impose restrictions on code execution and on access to resources. Security configuration settings are recorded in the properties file `${java.home}/conf/security/java.policy`, where the environment variable `java.home` is the root directory of a Java runtime image, or a directory where the JDK is installed on a given machine. This configuration file contains properties defined as name-value pairs that describe general security settings, references to certificate keystore files, and references to other security policy files that can be found in other locations.

Here is an example of a configuration in the `java.security` properties file to reference policy files:

Click here to view code image

```
policy.url.1=file:${java.home}/conf/security/java.policy
policy.url.2=file:/anypath/java.policy
```

These Java security policy descriptors configure security restrictions and permissions. These restrictions and permissions define access to resources and permissions to execute code for a specific codebase. The term *codebase* describes a location where Java code is placed, such as a directory or a URL, but more typically JAR archives. Each security policy is defined as a *grant* that is associated with a specific digital signature and allows the origin of the code to be authenticated.

Here is an example of a `java.policy` file structure:

Click here to view code image

```
keystore "xyz.keystore";
grant codeBase "file:/application.jar" signedBy "abc" {
  permission java.net.SocketPermission "localhost:8080", "listen";
  permission java.io.FilePermission "/FileSystemPath", "read, write";
};
```

The example above defines a keystore description and specifies a number of permissions. The keystore file is a secure store that contains keys and certificates. It is used to look up the public keys and associate digital signatures with a given codebase. Keystores are created and maintained using a keytool utility.

A grant is configured for a specific codebase given by the location of a JAR file containing classes to which this grant should be applied. In other words, classes within this archive will be allowed to perform restricted actions described within this grant. To make sure this is a genuine JAR file, it can be signed using a `signedBy` property that references a relevant digital signature alias from the keystore.

In the grant specification, this policy descriptor defines a number of permissions. One permission allows classes contained within the codebase to listen on a certain address given by the host and the specified port. Another permission allows read and write access to a specific file system path. The exact nature of the permissions depends on the permission type, such as a socket or file permission used in this example. Custom permissions can also be created by extending the `java.security .BasicPermission` class.

Once security policies are defined, they can be verified.

Click here to view code image

```
SocketPermission socketPermission
        = new SocketPermission("localhost:8080", "listen");     // (1)
FilePermission filePermission
        = new FilePermission("/FileSystemPath", "read, write"); // (2)
try {
  AccessController.checkPermission(socketPermission);           // (3)
  AccessController.checkPermission(filePermission);             // (4)
  // (5) ...
} catch(AccessControlException e) {
  // (6) ...
}
```

The numbered comments below correspond to the numbered lines in the code:

(1)–(2) A number of permission objects can be initialized to match permissions configured through the security policy descriptor.

(3)–(4) The `AccessController` class is used to validate whether such permissions were indeed granted to the given class. This is determined based on the class location within a codebase referenced by the relevant grant. The method `checkPermission()` throws an `AccessControlException` if corresponding permission was not granted.

(5) Once permissions are verified, the program can proceed to perform restricted actions.

(6) Handle exceptions that are thrown when requested access does not match permissions configured by the policies.

Executing Privileged Code

There are cases when it is not known which specific permissions must be verified—for example, when the program logic is supplied before the actual permission configuration has been created, or when such a configuration is expected to change in the future. In these cases, the program can use the `AccessController.doPrivileged()` method to execute the computation with privileges enabled. This method accepts an object that implements the `PrivilegedAction` or the `PrivilegedExceptionAction` interface, depending on whether this computation can potentially throw a checked exception.

Click here to view code image

```
String content = AccessController.doPrivileged(
    new PrivilegedAction<String>() {
      @Override
    public String run() {                                         // (1)
      String result = null;
      try {
        result = Files.readString(Path.of("/FileSystemPath")); // (2)
      } catch (IOException ex) {
        // ...                                                    (3)
      }
      return result;
    }
  });
```

The numbered comments below correspond to the numbered lines in the code:

1. The implementation of the `PrivilegedAction` interface must override the `run()` method.
2. The implementation of the `run()` method contains logic that has access to restricted resources.
3. Any checked exceptions thrown must be handled within the `run()` method. If checked exceptions are to propagate outside of the `run()` method, then the `PrivilegedExceptionAction` interface should be implemented instead.

The method `doPrivileged()` is overloaded and can accept additional parameters, including specific permissions that must be satisfied for the privileged action to succeed.

26.4 Additional Security Guidelines

Attention is drawn to some selected security guidelines in this section.

Accessing the File System

When accessing the file system or performing I/O operations, consider the following recommendations:

- Remove redundant elements from the path and convert it to a canonical form using the methods `normalize()` and `toRealPath()` to guard against directory traversal attacks, such as attempts to guess the directory structure by using relative paths (§21.3, p. 1297).
- Verify the file sizes and lengths of I/O streams before attempting to process information.
- Monitor I/O operations to detect excessive use of resources and terminate operations that process excessive amounts of data.
- Release resources as soon as possible, and terminate and time out long operations.

Deserialization

Be mindful of deserialization (§20.5, p. 1261), as it can create objects that are beyond the control of the application and that can bypass normal constructor invocations. Essentially, the deserialization process circumvents normal secure object creation, and thus must be considered a potential risk factor, especially if a serialized object arrived from an untrusted source. Thus it is a good idea to validate such objects as soon as they are created.

Class Design

Another important set of guidelines is related to class design.

- Always strive to enforce tight encapsulation using the most restrictive access permissions possible, also known as *principle of least privilege* (PoLP). Consider utilizing the Java Platform Module System for further reinforcement of encapsulation (§19.3, p. 1173), including restrictions imposed on the use of reflection (§19.8, p. 1191).
- Make objects immutable (§6.7, p. 356). Don't forget that even though a variable referencing an object can be marked as `final`, it does not mean that the object itself is immutable. It may be a good idea to create cloned object replicas of otherwise mutable objects to avoid unsafe memory mutable operations.
- When extending classes and overriding methods, carefully consider the semantics of the superclass methods to override. Also, remember that any modification made to a superclass, such as changing method implementation or introducing additional methods, may have a cascading effect on all its subclasses in the inheritance hierarchy (§5.1, p. 191).
- Always mark non-`private` classes and methods as `final` if they are not intended to be extended or overridden (§5.5, p. 225).
- Use factory methods to validate values before invoking a constructor to actually create an object.
- Avoid invoking overridden methods from constructors (§10.9, p. 555).
- Declare constructors as `private` if the class should not be instantiated.

Finally, consider never disabling bytecode verification with command-line options such as `-Xverify:none` or `-noverify` that protect bytecode against tampering and dangerous behavior.

Review Questions

26.1 Which of the following can prevent DoS attacks?

Select the one correct answer.

a. Obfuscating sensitive data
b. Sanitizing input values
c. Terminating recursive data references
d. Encapsulating code

26.2 An input parameter containing executable code is an example of which of the following?

Select the one correct answer.

a. Lack of encapsulation
b. Mutable code
c. Code corruption
d. Code injection

26.3 Given the following code:

Click here to view code image

```
public class RQ3 {
  public static void main(String[] args) {
    int z = Math.addExact(Integer.MAX_VALUE, 1);
    System.out.println(z);
  }
}
```

What is the result?

Select the one correct answer.

a. `2147483647`
b. `-2147483648`
c. `2147483648`
d. An `ArithmeticException`

26.4 Which of the following describes the meaning of the number 256 in the name of the SHA-256 algorithm?

Select the one correct answer.

a. A length of a public key used for encryption
b. A length of a private key used for encryption
c. A length of the resulting hash value

d. A length of the input value

26.5 What is the purpose of Java security policies?

Select the one correct answer.

a. To set restrictions on code execution and on access to resources

b. To set POSIX permissions for file system resources

c. To restrict access to data via security methods

d. To prevent code injections

26.6 Which of the following statements is true about privileged code execution?

Select the one correct answer.

a. The method `PrivilegedAction.run()` is expected to throw a checked exception if security constraints are violated.

b. The method `AccessController.doPrivileged()` can handle checked exceptions thrown by the `PrivilegedAction.run()` method.

c. Checked exceptions must be handled within the `PrivilegedAction.run()` method.

d. Checked exceptions thrown by the `PrivilegedAction.run()` method can be propagated out of the `AccessController.doPrivileged()` method.

Appendix A

Taking the Java SE 17 and Java SE 11 Developer Exams

Please note that all information presented in this appendix is valid as of November 2022. It is imperative that readers regularly visit the websites mentioned in this appendix, as Oracle is known to change practical information regarding the exam and the exam objectives intermittently.

The primary focus of this book is the *Java SE 17 Developer Exam* (*1Z0-829*) and the *Java SE 11 Developer Exam* (*1Z0-819*). Pertinent information about these two exams can be found here:

Click here to view code image

```
https://education.oracle.com/java-se-17-developer/pexam_1Z0-829
https://education.oracle.com/java-se-11-developer/pexam_1Z0-819
```

For authoritative information about the certification paths for Java SE, consult the following web page:

Click here to view code image

```
https://education.oracle.com/oracle-certification-path/product_267
```

Appendix B, p. 1623, and Appendix C, p. 1629, contain specific information about the objectives for the *Java SE 17 Developer* exam and the *Java SE 11 Developer* exam, respectively.

A.1 Preparing for the Exam

The goal of the exams is to test practical knowledge and usage of the Java programming language and specific APIs. The exams require a thorough understanding of both the syntax and the semantics of the language, including the use of its core APIs. The exams cover a wide variety of topics, as defined in the exam objectives (Appendix B, Appendix C). Central to the exams are language constructs, usage of the core API, and specific topics, with heavy emphasis on interpreting code scenarios.

The need for real-world experience prior to taking the exam cannot be stressed enough. It requires very thorough preparation to pass the exam without having some actual Java programming experience. This book is intended to be used as a study guide: to encourage readers to try out what they have learned every step of the way, and to test their newly acquired knowledge using the review questions provided in every chapter.

Experimenting with the examples in this book will give the reader a much better chance of passing the exam. Numbered examples in the book are complete Java programs, the source code for which is available on the companion book website. Whether one uses the tools in the JDK or an IDE (integrated development environment), it is the hands-on programming that is important. That said, tools in the JDK are preferred as preparation for the exam, as there is less reliance on programming support provided by an IDE.

When the reader feels ready, they should test their skills on the mock exam for Java SE 17 provided in Appendix E. This will give the reader an indication of how well they are prepared for the exam, and which topics need further study. The structure of the book should make it easy for the reader to focus on single topics, if necessary.

The exam is considered to be difficult, and it requires a fair amount of studying on the part of the candidate. Even seasoned Java programmers should invest some time in preparing for the exam. Simply having real-world experience is not enough to pass the exam.

It is highly recommended that exam candidates devise a study plan, incorporating the activities we mentioned previously. Disruptions in the study plan should be avoided, as this can result in revising the material already covered when you pick up the thread. The exam should be scheduled immediately after the study period, when the momentum from the preparation is at its maximum.

We also highly recommend joining an online Java certification community, such as Code Ranch (`www.coderanch.com`), which has dedicated active forums for the different Java certifications. All things Java are discussed in a friendly manner around the bonfire at this ranch, where greenhorns are especially welcomed. The forum for Java Programmer Certification can be found here:

Click here to view code image

```
https://coderanch.com/f/24/java-programmer-OCPJP
```

The following website provides more practice tests for OCP Java SE Developer certification at a nominal price:

Click here to view code image

```
https://enthuware.com/java-certification-mock-exams/oracle-certified-profes-
sional/ocp-java-17-exam-1z0-829
```

For answers to technical questions on Java, consult the following website:

```
https://stackoverflow.com
```

A.2 Registering for the Online Proctored Exam

The best way to find all pertinent up-to-date information for taking an exam is to visit the exam website, which provides all the relevant links on buying an exam, registering for the exam, and scheduling the exam.

A *certified Oracle account* is required to log on to the *Oracle CertView* certification portal, where pertinent information about the exam can be found. These steps can be accomplished at:

```
https://certview.oracle.com
```

Obtaining an Oracle Exam Attempt

An *exam attempt* (previously called a *certification voucher*) to take the exam is sold by Oracle.

Click here to view code image

```
https://education.oracle.com/buy-exam
```

Be sure to obtain an exam attempt for *Oracle Technology Exams*, which covers Java certification exams. Credit card information is usually required to arrange payment. The cost of the exam attempt may vary depending on the country you live in and on any offers that might apply at the time. For US residents, the regular price is around $245 at the time of writing this book.

Note that the exam attempt has an expiration date, which is usually six months after it is acquired. Oracle will not replace lost or expired exam attempts, neither will they offer refunds for unused exam attempts.

Scheduling for the Online Proctored Exam

After obtaining the exam attempt, candidates can schedule to take the online proctored exam when they are ready at the *Oracle MyLearn* portal. Be sure to schedule for the right exam. Candidates can reschedule an exam without penalty up to 48 hours before the appointment time.

```
https://mylearn.oracle.com
```

Candidate Agreement

Before taking the exam, candidates will have to agree to the terms and conditions of the *Oracle Certification Program Candidate Agreement* found at this site:

Click here to view code image

```
https://education.oracle.com/file/general/ocp_candidate_agreement.pdf
```

After Taking the Exam

After taking the exam, candidates will receive an email from Oracle within 30 minutes informing them that their exam results are available in CertView. Candidates who pass the exam will receive an email informing them that an eCertificate is available in CertView. Instructions for requesting a printed copy of the certificate are included in this email.

Candidates who fail the exam can register to retake the exam after a 14-day waiting period.

A.3 How the Online Proctored Exam Is

Conducted

The English-language exam is conducted as an online proctored exam. Comprehensive instructions for the online exam can be found at the *Oracle MyLearn* portal.

Requirements for the Online Proctored Exam

Specific requirements are stipulated at the Oracle MyLearn portal for the computer equipment, required ID proofs, and test space in order to take the online exam. A proctor will ensure that these requirements are met before the exam begins and during the exam.

Well before the exam, it is important that candidates familiarize themselves with and prepare for the exam, including making sure all exam requirements are met. Candidates should consult the Oracle MyLearn portal for the requirements that are in effect at any given time. The following is a short summary of these requirements at the time of writing this book.

- *Computer equipment*

  Minimum requirements are spelled out for computer equipment (personal desktop/laptop) with regard to Internet connectivity, audio, and webcam capabilities. Candidates will need to prepare their equipment in advance so that it meets the minimum requirements.

- *Proof of identity*

  The Oracle MyLearn portal states the exact requirements regarding two forms of government-issued identifications that may be required to take the exam.

- *Testing space*

  Apart from checking in 30 minutes before the exam and taking it in a quiet, distraction-free room, there are strict requirements as to what is *not* allowed during the exam. Considering the mode of exam delivery, these restrictions are not surprising as they create a test space where the only communication allowed is that between the candidate and the exam delivery software.

The Exam Program

The computer program used to conduct the exam will select a set of questions at random, and present them through a graphical user interface. The interface is designed in such a way that candidates are able to move back and forth through the questions for reviewing purposes. Questions can be temporarily left unanswered, and the candidate can return to them later. The candidate can also mark questions for later review during the exam.

Before the exam starts, the candidate is allowed a test run with the test interface. A demo test that has nothing to do with the Java exam is used. Its sole purpose is to acquaint the candidate with the program being used to conduct the exam.

Utilizing the Allotted Time

The exam consists of a fixed number of questions that must be answered within the allocated time. As candidates have limited time to answer each question, they cannot afford to get stuck on a question. If the answer does not become apparent within a reasonable time, it is advisable to move on to the next question. Time permitting, it is possible to return to the unanswered questions later. It is important to answer *all* questions, however a wrong answer and a blank answer carry the same penalty: loss of points. Therefore, it is better to guess an answer and hope that it is correct rather than leaving blank. The process of elimination can sometimes be useful in narrowing down the answer to a question. Eliminating obvious incorrect choices increases the chances of arriving at the right answer.

An experienced Java programmer used to taking exams should be able to complete the exam within the allotted time. Any remaining time is best used to review one’s answers.

The Exam Result

The exam results are presented on the screen after the exam. Afterwards, candidates should log on to the CertView certification portal to see details about their performance:

- An indication of whether they passed or failed.
- The total score. Only the scored questions on the exam contribute to the final score. All the scored questions are weighted equally, and the score is calculated based on the percentage of correct answers. No credit is given for

partially correct answers for the scored questions.

- Indications on how well the candidate did on each category within the objectives. Candidates who fail the exam should pay close attention to this information. If the candidate is planning to retake the exam, it may give a good indication of which topics need closer attention.

The result will not divulge which questions were answered correctly.

A.4 The Questions

Assumptions About the Exam Questions

Candidates should be aware of certain assumptions made about the questions on the *Java SE 17 Developer Exam*. In the following list, we provide a short explanation of these assumptions.

- *Missing package and import statements*

  Unless explicitly provided, stated, or referred to in the question, assume that the code is in the same package/ module and all necessary import statements are given.

- *No file or directory pathnames for classes*

  In this case, assume that either all classes are in the same file or each class is in a separate file and these files are in the same directory.

- *Unintended line breaks*

  Line breaks that make the code lines appear to be wrapped unintentionally should be ignored, and the code should be assumed to compile without errors.

- *Code fragments*

  Assume that the necessary context exists to compile and execute the code, if such context is not explicitly specified by the question.

- *Descriptive comments*

  Such comments should be taken at their face value, providing the intent described in the comment.

Types of Questions Asked

Most of the questions follow a common format that requires candidates to apply their knowledge in a special way.

- Analyzing program code. The question provides some source code, and asks a specific question pertaining to the code. Will running the program provide the expected result? What will be written to the standard output when the program is run? Will the code compile? Choosing a code snippet to insert in the given code in order for it to exhibit specific behavior.
- Identifying true or false statements.

When analyzing program code, it is useful to try to apply the same rules as the compiler: examining the exact syntax used, rather than making assumptions on what the code tries to accomplish.

The wording of the questions is precise, and the responses selected in multiple-choice questions are likewise expected to be precise. This often causes the test to be perceived as fastidious. Close attention should be paid to the wording of the responses in a multiple-choice question.

None of the questions is intentionally meant to be a trick question. Exam questions have been reviewed by both Java experts and language experts to remove as much ambiguity from their wording as possible.

Since the program used in the exam will select and present the questions in a random fashion, there is no point in trying to guess the form of the questions. The order of the answers in multiple-choice questions has been randomized, and therefore, has no significance.

Types of Answers Expected

All exam questions are multiple choice. The correct number of alternatives to select is designated in the question, and all must be selected for the question as a whole to be considered correctly answered.

There should be no problem identifying which form of answer each question requires. The wording of the questions will indicate this, and the software used will present the candidate with an input method corresponding to the form of answer expected.

For multiple-choice questions, the program will ask the candidate to select a specific number of answers from a list. Where a single correct answer is expected, radio buttons will allow selection of only one answer. The most appropriate response should be selected.

In questions where all appropriate responses should be selected, check boxes will allow the selection of each response individually. In this case, all choices should be considered on their own merits; that is, responses should not be weighed against each other. It can be helpful to think of each of the choices as an individual true–false question.

Topics Covered by the Questions

Topics covered by the exam are basically derived from the set of exam objectives defined by Oracle. Not all topics may appear on an actual exam session, due to the limited number of questions comprising the exam. The objectives for the *Java SE 17* and *11 Developer* exams are included in Appendix B and Appendix C, respectively, with references to where the topics are covered in the book.

Many of the questions will require intimate knowledge of core Java APIs. This book covers the most important classes and methods of the Java SE Platform API, but it does not go as far as listing every member of every class. The Java SE Platform API documentation should be consulted, as it is essential that candidates familiarize themselves with the relevant parts of the API documentation.

Appendix B

Exam Topics: Java SE 17 Developer

Please note that all information presented in this appendix is valid as of November 2022. It is imperative that readers visit the exam website mentioned in this appendix regularly while preparing for the exam, as Oracle is known to change the exam objectives intermittently.

Exam Name: *Java SE 17 Developer*
Exam Number: *1Z0-829*
Certification: *Oracle Certified Professional: Java SE 17 Developer*
Duration: *90 minutes*
Number of Questions: *50*
Passing Score: *68%*
Exam Format: *Multiple Choice*
Exam Price: *$245*

Candidates must pass the *Java SE 17 Developer* exam in order to qualify as an *Oracle Certified Professional: Java SE 17 Developer*. Pertinent information about this exam can be found at:

Click here to view code image

```
https://education.oracle.com/java-se-17-developer/pexam_1Z0-829
```

The web page also lists the exam topics defined by Oracle. The topics are organized in *sections*, and each section is *reproduced verbatim* in this appendix. For each section, we have provided references to where in the book the exam topics (we call them *objectives*) in the section are covered. In addition, the extensive index at the end of the book can also be used to look up specific topics.

General information about taking the exam can be found in Appendix A, p. 1615. Oracle has also specified certain important assumptions about the exam questions, which can also be found in Appendix A, p. 1619. In addition to the exam objectives and the assumptions, the exam web page lists a number of topics that the candidate is expected to know, but the exam will not include any direct questions on these topics. We have classified them as *supplementary topics* (p. 1627).

Section 1: Handling da and boolean values

Chapters: 2, 8, 17

[1.1] Use primitives and wrapper classes including Math API, parentheses, type promotion, and casting to evaluate arithmetic and boolean expressions

§2.2, p. 41 to §2.19, p. 92 §8.3, p. 429 §8.6, p. 478

[1.2] Manipulate text, including text blocks, using String and StringBuilder classes

§8.4, p. 439 §8.5, p. 464

[1.3] Manipulate date, time, duration, period, instant and time-zone objects using Date-Time API

§17.1, p. 1024 to §17.7, p. 1072

Section 2: Controlling

Chapter 4

[2.1] Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements

§4.1, p. 152 to §4.13, p. 184

Section 3: Utilizing Ja Approach

Chapters: 3, 5, 6, 9, 10, 14

[3.1] Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection

§9.1, p. 491 to §9.6, p. 521 §10.1, p. 533 to §10.4, p. 537

[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers

§3.1, p. 99 to §3.8, p. 112 §5.14, p. 299 §10.5, p. 540 to §10.9, p. 555

[3.3] Implement overloading, including var-arg methods

§3.6, p. 108 §3.11, p. 136 §5.1, p. 202

[3.4] Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable

§3.13, p. 142 §6.1, p. 324 §6.6, p. 352 §6.7, p. 356

[3.5] Implement inheritance, including abstract and sealed classes. Override methods, including that of Object class. Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching

§5.1, p. 191 to §5.4, p. 218 §5.11, p. 269 §5.12, p. 278 §5.15, p. 311 §14.1, p. 743 to §14.3, p. 753

[3.6] Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods

§5.6, p. 237

[3.7] Create and use enumerations with fields, methods and constructors	§5.13, p. 287
Section 4: Handling Ex	**Chapter 7**
[4.1] Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions	§7.2, p. 375 §7.3, p. 375 §7.6, p. 397 §7.7, p. 407
Section 5: Working wit Collections	**Chapters: 3, 11, 12, 14, 15**
[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements	§3.9, p. 117 §11.1, p. 565 §11.13, p. 623 §12.1, p. 644 §12.8, p. 662 §14.4, p. 761 §14.5, p. 769 §15.1, p. 783 §15.12, p. 864
Section 6: Working wit expressions	**Chapters: 13, 16**
[6.1] Use Java object and primitive Streams, including lambda expressions implementing functional interfaces, to supply, filter, map, consume, and sort data	§13.1, p. 675 §13.14, p. 733 §16.3, p. 884 §16.7, p. 946
[6.2] Perform decomposition, concatenation and reduction, and grouping and partitioning on sequential and parallel streams	§16.7, p. 946 §16.8, p. 978 §16.9, p. 1009
Section 7: Package and use the Java Platform Module System	**Chapter 19**
[7.1] Define modules and their dependencies, expose module content including for reflection. Define services, producers, and consumers	§19.1, p. 1163 §19.5, p. 1179 §19.8, p. 1191 §19.9, p. 1196
[7.2] Compile Java code, produce modular and non-modular jars, runtime images, and implement migration using unnamed and automatic modules	§19.6, p. 1186 §19.7, p. 1189 §19.10, p. 1204 §19.14, p. 1218
Section 8: Manage con	**Chapters: 16, 22, 23**
[8.1] Create worker threads using Runnable and Callable, manage the thread lifecycle, including automations provided by different Executor services and concurrent API	§22.1, p. 1367 §22.5, p. 1408 §23.2, p. 1423
[8.2] Develop thread-safe code, using different locking mechanisms and concurrent API	§23.4, p. 1451 §23.7, p. 1482
[8.3] Process Java collections concurrently including the use of parallel streams	§16.9, p. 1009 §23.7, p. 1482
Section 9: Use Java I/C	**Chapters: 20, 21**
[9.1] Read and write console and file data using I/O Streams	§20.1, p. 1233 §20.4, p. 1256
[9.2] Serialize and de-serialize Java objects	§20.5, p. 1261
[9.3] Create, traverse, read, and write Path objects and their properties using java.nio.file API	§21.1, p. 1287 §21.8, p. 1345

Section 10: Access data	**Chapter 24**
[10.1] Create connections, create and execute basic, prepared and callable statements, process query results and control transactions using JDBC API	§24.1, p. 1512 to §24.8, p. 1545
Section 11: Implement	**Chapter 18**
[11.1] Implement localization using locales, resource bundles, parse and format messages, dates, times, and numbers including currency and percentage values	§18.1, p. 1096 to §18.7, p. 1139
Section 12: Supplemen	**Chapters: 11, 25**
[12.1] Understand the basics of Java Logging API.	See *Appendix C, p. 1747*.
[12.2] Use Annotations such as Override, FunctionalInterface, Deprecated, SuppressWarnings, and SafeVarargs.	*Standard annotations are covered in §25.5, p. 1577, and are used throughout the book, see also the index.*
[12.3] Use generics, including wildcards.	Generics are covered in *Chapter 11, p. 563*.

Appendix C

Exam Topics: Java SE 11 Developer

Please note that all information presented in this appendix is valid as of November 2022. It is imperative that readers visit the exam website mentioned in this appendix regularly while preparing for the exam, as Oracle is known to change the exam objectives intermittently.

Exam Name: *Java SE 11 Developer*
Exam Number: *1Z0-819*
Certification: *Oracle Certified Professional: Java SE 11 Developer*

Duration: *90 minutes*
Number of Questions: *50*
Passing Score: *68%*
Exam Format: *Multiple Choice*
Exam Price: *$245*

Candidates must pass the *Java SE 11 Developer* exam in order to qualify as an *Oracle Certified Professional: Java SE 11 Developer*. Pertinent information about this exam can be found at:

Click here to view code image

```
https://education.oracle.com/java-se-11-developer/pexam_1Z0-819
```

The web page also lists the exam topics defined by Oracle. The topics are organized in *sections*, and each section is *reproduced verbatim* in this appendix. For each section, we have provided references to where in the book the exam topics (we call them *objectives*) in the section are covered. In addition, the extensive index at the end of the book can also be used to look up specific topics. General information about taking the exam can be found in Appendix A, p. 1615. Oracle has also specified certain important assumptions about the exam questions, which can also be found in Appendix A.

Section 1: Working wit	Chapters: 2, 3, 8
[1.1] Use primitives and wrapper classes, including, operators, the use of parentheses, type promotion and casting	§2.2, p. 41; §2.19, p. 92; §8.3, p. 429
[1.2] Handle text using String and StringBuilder classes	§8.4, p. 439; §8.5, p. 464
[1.3] Use local variable type inference, including as lambda parameters	§3.13, p. 142

Section 2: Controlling	Chapter 4
[2.1] Create and use loops, if/else, and switch statements	§4.1, p. 152; §4.8, p. 176

Section 3: Java Object	Chapters: 3, 5, 6, 9, 10, 13
[3.1] Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)	§9.1, p. 491; §9.6, p. 521; §10.1, p. 533; §10.4, p. 537
[3.2] Define and use fields and methods, including instance, static and overloaded methods	§3.2, p. 99; §3.8, p. 112
[3.3] Initialize objects and their members using instance and static initialiser statements and constructors	§3.4, p. 102; §3.7, p. 109; §10.5, p. 540; §10.9, p. 555
[3.4] Understand variable scopes, applying encapsulation and make objects immutable	§6.1, p. 324; §6.6, p. 352; §6.7, p. 356
[3.5] Create and use subclasses and superclasses, including abstract classes	§5.1, p. 191; §5.4, p. 218
[3.6] Utilize polymorphism and casting to call methods, differentiating object type versus reference type	§5.11, p. 269; §5.12, p. 278
[3.7] Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods	§5.6, p. 237; §13.1, p. 675
[3.8] Create and use enumerations	§5.13, p. 287

Section 4: Exception Ha	Chapter 7
[4.1] Handle exceptions in the Java program by using try/catch/finally clauses, try-with-resource, and multi-catch statements	§7.3, p. 375; §7.6, p. 397; §7.7, p. 407
[4.2] Create and use custom exceptions	§7.2, p. 375

Section 5: Working wit Collections	Chapters: 3, 11, 12, 14, 15
[5.1] Use generics, including wildcards	§11.1, p. 565; §11.13, p. 623
[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods	§3.9, p. 117; §12.1, p. 644; §12.8, p. 662; §15.1, p. 783; §15.9, p. 840
[5.3] Sort collections and arrays, using Comparator and Comparable interfaces	§14.4, p. 761; §14.5, p. 769; §15.5, p. 810; §15.10, p. 845; §15.11, p. 858; §15.12, p. 865

Section 6: Working wit expressions

Chapters: 13, 16

[6.1] Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package	*§13.1, p. 675 to §13.14, p. 733*
[6.2] Use Java Streams to filter, transform and process data	*§16.3, p. 884 to §16.7, p. 946*
[6.3] Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams	*§16.7, p. 946, §16.8, p. 978, §16.9, p. 1009*

Section 7: Java Platfor

Chapter 19

[7.1] Deploy and execute modular applications, including automatic modules	*§19.2, p. 1164, §19.6, p. 1186, §19.7, p. 1189, §19.10, p. 1204 to §19.13, p. 1214*
[7.2] Declare, use, and expose modules, including the use of services	*§19.3, p. 1168, §19.4, p. 1177, §19.5, p. 1179, §19.8, p. 1191, §19.9, p. 1196*

Section 8: Concurrency

Chapters: 22, 23

[8.1] Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API	*§22.1, p. 1371 to §23.2, p. 1423*
[8.2] Develop thread-safe code, using different locking mechanisms and java.util.concurrent API	*§23.4, p. 1451 to §23.7, p. 1482*

Section 9: Java I/O API

Chapters: 20, 21

[9.1] Read and write console and file data using I/O Streams	*§20.1, p. 1233 to §20.4, p. 1256*
[9.2] Implement serialization and deserialization techniques on Java objects	*§20.5, p. 1261*
[9.3] Handle file system objects using java.nio.file API	*§21.1, p. 1287 to §21.8, p. 1345*

Section 10: Secure Cod Application

Chapter 26

[10.1] Develop code that mitigates security threats such as denial of service, code injection, input validation and ensure data integrity	*§26.1, p. 1600, §26.2, p. 1602*
[10.2] Secure resource access including filesystems, manage policies and execute privileged code	*§26.3, p. 1608, §26.4, p. 1610*

Section 11: Database A

Chapter 24

[11.1] Connect to and perform database SQL operations, process query results using JDBC API	*§24.1, p. 1512 to §24.8, p. 1545*

Section 12: Localizatio

Chapter 18

[12.1] Implement Localization using Locale, resource bundles, and Java APIs to parse and format messages, dates, and numbers	*§18.1, p. 1096 to §18.7, p. 1139*

Section 13: Annotation

Chapter 25

[13.1] Create, apply, and process annotations	*§25.1, p. 1557 to §25.6, p. 1587*

Appendix D

Annotated Answers to Review Questions

1 Basics of Java Programming

1.1 *(c)*

A method is an operation defining a particular behavior of an abstraction. Java implements abstractions using classes that have properties and behaviors. Behaviors are defined by the operations of the abstraction.

1.2 *(b)*

An object is an instance of a class. Objects are created from classes that implement abstractions. The objects that are created are concrete realizations of those abstractions. An object is neither a reference nor a variable.

1.3 *(b)*

(2) is the first line of a constructor declaration. A constructor in Java is declared like a method that does not return a value. It has the same name as the class name, but it does not specify a return type and therefore does not return a value. (1) is the header of a class declaration, (3) is the first statement in the constructor body, and (4), (5), and (6) are instance method declarations.

1.4 *(b) and (f)*

Two objects are created and three references are declared by the code. Objects are normally created by using the `new` operator. The declaration of a reference creates a variable regardless of whether a reference value is assigned to it or not.

1.5 (d)

An instance member is a field or an instance method. These members belong to all instances of the class. Members that are not explicitly declared `static` in a class declaration are instance members.

1.6 (c)

An object communicates with another object by calling an instance method of the other object, passing and receiving any information that might be necessary.

1.7 (d) and (f)

Given the declaration `class B extends A {...}`, we can conclude that class `B` extends class `A`, class `A` is the superclass of class `B`, class `B` is a subclass of class `A`, and class `B` inherits from class `A`, which means that objects of class `B` inherit the field `value1` from class `A`.

1.8 (d)

The compiler supplied with the JDK is named `javac`. The names of the source files to be compiled are listed on the command line after the command `javac`. (c) will compile and execute the program, but will not create a class file.

1.9 (a)

Java programs are executed by the Java Virtual Machine (JVM). In the JDK, the command `java` is used to start the execution by the JVM. The `java` command requires the name of a class that has a valid `main()` method. The JVM starts the program execution by calling the `main()` method of the given class. The exact name of the class should be specified, and not the name of the class file—that is, the `.class` extension in the class file name should not be specified. Since it is specified that the source file is compiled creating a class file, (c) would not work.

1.10 (a) and (d)

The file with a single-file source-code program can contain more than one class declaration and the first class declaration must provide a valid `main()` method. Such a program cannot access previously compiled user-defined classes, only those in the standard library. It cannot consist of multiple files obviously, but program arguments can be supplied on the command line.

1.11 *(e)*

(a) is incorrect because the JVM must be compatible with the Java Platform on which the program was developed.
(b) is incorrect because the JIT feature of the JVM translates bytecode to machine code.
(c) is incorrect because other languages, like Scala, also compile to bytecode and can be executed by the JVM.
(d) is incorrect because a Java program can only create objects, but destroying objects is at the discretion of the automatic garbage collector.

2 Basic Elements, Primitive Data Types, and Operators

2.1 *(e)*

Everything from the start sequence (`/*`) of a multiple-line comment to the first occurrence of the end sequence (`*/`) of a multiple-line comment is ignored by the compiler. Everything from the start sequence (`//`) of a single-line comment to the end of the line is ignored by the compiler. In (e), the multiple-line comment ends with the first occurrence of the end sequence (`*/`), leaving the second occurrence of the end sequence (`*/`) unmatched.

2.2 *(d)*

An assignment statement is an expression statement. The value of the expression statement is the value of the expression on the right-hand side. Since the assignment operator is right associative, the statement `a = b = c = 20` is evaluated as follows: `(a = (b = (c = 20)))`. This results in the value `20` being assigned to `c`, then the same value being assigned to `b` and finally to `a`. The program will compile and print `20` at runtime.

2.3 *(c)*

In an assignment statement, the reference value of the source reference is assigned to the destination reference. Assignment does not create a copy of the object denoted by the source reference. After the assignment, both references denote the same object—that is, they are aliases.

The variables `a`, `b`, and `c` are references of type `String`. The reference value of the `"cat"` object is first assigned to `a`, then to `b`, and later to `c`. Just before the print statement, `a` denotes `"dog"`, whereas both `b` and `c` denote `"cat"`. The program prints the string denoted by `c`—that is, `"cat"`.

2.4 *(a), (d), and (e)*

A binary expression with any floating-point operand will be evaluated using floating-point arithmetic. Expressions such as `2/3`, where both operands are integers, will use integer arithmetic and evaluate to an integer value. In (e), the result of `(0x10 * 1L)` is promoted to a floating-point value.

2.5 *(b)*

The `/` operator has higher precedence than the + operator. This means that the expression is evaluated as `((1/2) + (3/2) + 0.1)`. The associativity of the binary operators is from left to right, giving `(((1/2) + (3/2)) + 0.1)`. Integer division results in `((0 + 1) + 0.1)`, which evaluates to 1.1.

2.6 *(b)*

The expression evaluates to `-6`. The whole expression is evaluated as `(((-(-1)) -((3 * 10) / 5)) - 1)` according to the precedence and associativity rules.

2.7 *(d)*

The expression `++k + k++ + + k` is evaluated as `((++k) + (k++)) + (+k)` → ((2) + (2) + (3)), resulting in the value `7`.

2.8 *(d)*

The types `char` and `int` are both integral. A `char` value can be assigned to an `int` variable since the `int` type is wider than the `char` type and an implicit widening conversion will be done. An `int` type cannot be assigned to a `char` variable because the `char` type is narrower than the `int` type. The compiler will report an error about a possible loss of precision at (4).

2.9 *(a)*

First, the expression `++i` is evaluated, resulting in the value `2`. Now the variable `i` also has the value `2`. The target of the assignment is now determined to be the element `array[2]`. Evaluation of the right-hand expression, `--i`, results in the value `1`.

The variable `i` now has the value `1`. The value of the right-hand expression `1` is then assigned to the array element `array[2]`, resulting in the array contents to become `{4, 8, 1}`. The program computes and prints the sum of these values—that is, `13`.

2.10 *(c) and (e)*

The remainder operator is not limited to integral values, but can also be applied to floating-point operands. Short-circuit evaluation occurs with the conditional operators (`&&`, `||`). The operators `*`, `/`, and `%` have the same level of precedence. The data type `short` is a 16-bit signed two's complement integer, thus the range of values is from `-32768` to `+32767`, inclusive. `(+15)` is a legal expression using the unary `+` operator.

2.11 *(a), (c), and (e)*

The `!=` and `^` operators, when used on boolean operands, will return `true` if and only if one operand is `true`, and `false` otherwise. This means that `d` and `e` in the program will always be assigned the same value, given any combination of truth values in `a` and `b`. The program will, therefore, print `true` four times.

2.12 *(b)*

The element referenced by `a[i]` is determined based on the current value of `i`, which is `0`—that is, the element `a[0]`. The expression `i = 9` will evaluate to the value `9`, which will be assigned to the variable `i`. The value `9` is also assigned to the array element `a[0]`. After execution of the statement, the variable `i` will contain the value `9`, and the array `a` will contain the values `9` and `6`. The program will print `9 9 6` when run.

2.13 *(c) and (d)*

Note that the logical and conditional operators have lower precedence than the relational operators. Unlike the `&` and `|` operators, the `&&` and `||` operators short-circuit the evaluation of their operands if the result of the operation can be determined from the value of the first operand. The second operand of the `||` operator in the program is never evaluated because the value of `t` remains `true`. All the operands of the other operators are evaluated. Variable `i` ends up with the value `3`, which is the first digit printed, and `j` ends up with the value `1`, which is the second digit printed.

2.14 *(b)*

Both `||` and `&&` are short-circuit conditional operators. In the conditional expression `(x < y || ++z > 4)` of the first `if` statement, since the first operand `x < y` evaluates to `true`, the second operand `++z > 4` is not evaluated, as the conditional operator is `||`. The `if` condition is `true` and the `if` block is executed, printing `a123`.

In the conditional expression `(x < y || ++z > 4)` of the second `if` statement, since the first operand `x < y` evaluates to `true`, the second operand `++z > 4` is evaluated, as the conditional operator is `&&`. The second operand is `false` (`4 > 4`); therefore, the `if` condition is `false` and the `if` block is not executed.

2.15 *(c), (e), and (f)*

In (a), the third operand has the type `double`, which is not assignment compatible with the type `int` of the variable `result1`. Blocks are not legal operands in the conditional operator, as in (b). In (c), the last two operands result in wrapper objects with type `Integer` and `Double`, respectively, which are assignment compatible with the type `Number` of the variable `number`. The evaluation of the conditional expression results in the reference value of an `Integer` object with value `20` being assigned to the `number` variable. All three operands of the operator are mandatory, which is not the case in (d). In (e), the last two operands are of type `int`, and the evaluation of the conditional expression results in an `int` value (`21`), whose text representation is printed. In (f), the value of the second operand is boxed into a `Boolean`. The evaluation of the conditional expression results in a string literal (`"i not equal to j"`), which is printed. The `println()` method creates and prints a text representation of any object whose reference value is passed as a parameter.

2.16 *(d)*

The condition in the outer conditional expression is `false`. The condition in the nested conditional expression is `true`, resulting in the value of `m1` (i.e., `20`) being printed.

3 Declarations

3.1 *(c)*

The local variable of type `float` will remain uninitialized.

Fields and static variables are initialized with a default value. An instance variable of type `int[]` is a reference variable that will be initialized with the `null` value. Local variables remain uninitialized unless explicitly initialized.

3.2 *(e)*

The program will compile. The compiler can figure out that the local variable `price` will always be initialized, since the value of the condition in the `if` statement is `true`. The two instance variables and the two static variables are all initialized to the respective default value of their type.

3.3 *(a) and (e)*

The first and the third pairs of methods will compile. The second pair of methods will fail to compile, since their method signatures do not differ. The compiler has no way of differentiating between the two methods. Note that the return type and the names of the parameters are not a part of the method signature. Both methods in the first pair are named `fly` and have a different number of parameters, thus overloading this method name. The methods in the last pair do not overload the method name `glide`, since only one method has that name. The method named `Glide` is distinct from the method named `glide`, as identifiers are case sensitive in Java.

3.4 *(b) and (e)*

A constructor can be declared `private`, but this means that this constructor can only be used within the class. Constructors need not initialize all the fields when a class is instantiated. A field will be assigned a default value if not explicitly initialized. A constructor is non-static, and as such it can directly access both the static and non-static members of the class.

3.5 *(c)*

A compile-time error will occur at (3), since the class does not have a constructor accepting a single argument of type `int`. The declaration at (1) declares a method, not a constructor, since it is declared as `void`. The method happens to have the same name as the class, but that is irrelevant. The class has the default constructor, since the class contains no constructor declarations. This constructor will be invoked to create a `MyClass` object at (2).

3.6 *(b)*

The keyword `this` can only be used in non-`static` code, as in non-`static` methods, constructors, and instance initializer blocks. Only one occurrence of each `static` variable of a class is created, when the class is loaded by the JVM. This occurrence is shared among all the objects of the class (and for that matter, by other clients). Local variables are only accessible within the local scope, regardless of whether the local scope is defined within a static context.

3.7 *(e)*

The `[]` notation can be placed both after the type name and after the variable name in an array declaration. Multidimensional arrays are created by constructing arrays that can contain references to other arrays. The expression `new int[4][]` will create an array of length 4, which can contain references to arrays of `int` values. The expression `new int[4][4]` will also create a two-dimensional array, but will in addition create four more one-dimensional arrays, each of length 4 and of the type `int[]`. References to each of these arrays are stored in the two-dimensional array. The expression `int[][4]` will not work, because the arrays for the dimensions must be created from left to right.

3.8 *(a), (c), and (d)*

The size of the array cannot be specified, as in (b) and (e). The size of the array is given implicitly by the initialization code. The size of the array is never specified in the declaration of an array reference. The size of an array is always associated with the array instance (on the right-hand side), not the array reference (on the left-hand side).

3.9 *(e)*

The array declaration is valid, and will declare and initialize an array of length 20 containing `int` values. All the values of the array are initialized to their default value of `0`. The `for(;;)` loop will print all the values in the array—that is, it will print `0` twenty times.

3.10 *(d)*

The program will print `0 false 0 null` when run. All the instance variables, and the array element, will be initialized to their default values. When concatenated with a string, the values are converted to their text representation. Notice that the `null` literal is converted to the string `"null"`, rather than throwing a `NullPointerException`.

3.11 (b)

Evaluation of the actual parameter `i++` yields `0`, and increments `i` to `1` in the process. The value `0` is copied into the formal parameter `i` of the method `addTwo()` during method invocation. However, the formal parameter is local to the method, and changing its value does not affect the value in the actual parameter. The value of the variable `i` in the `main()` method remains `1`.

3.12 (d)

The variables `a` and `b` are local variables of type `int`. When these variables are passed as arguments to another method, the method receives copies of the primitive values in the variables. The actual variables are unaffected by operations performed on the copies of the primitive values within the called method. The variable `bArr` contains a reference value that denotes an array object containing primitive values. When the variable is passed as a parameter to another method, the method receives a copy of the reference value. Using this reference value, the method can manipulate the object that the reference value denotes. This allows the elements in the array object referenced by `bArr` to be accessed and modified in the method `inc2()`.

3.13 (c)

In (a) and (b), the arguments are encapsulated as elements in the implicitly created array that is passed to the method. In (c), the `int` array object itself is encapsulated as an element in the implicitly created array that is passed to the method. (a), (b), and (c) are fixed arity calls. Note that `int[]` is not a subtype of `Object[]`. In (d), (e), and (f), the argument is a subtype of `Object[]`, and the argument itself is passed without the need for an implicitly created array—that is, these are fixed arity method calls. However, in (d) and (e), the compiler issues a warning that both fixed arity and variable arity method calls are feasible, but chooses fixed arity method calls.

3.14 (b)

Local variable type inference with `var` is not allowed in a multiple-declaration statement, as at (2).

3.15 (d), (e), and (f)

The restricted keyword `var` cannot be used as a return type or as the type of a formal parameter, ruling out (a), (b), and (c).

The signature of the method call `divide(int, int)` is assignment compatible with the method signatures `divide(int, int)`, `divide(int, double)`, and `divide(double, int)` in (d), (e), and (f), respectively. The `double` value of the expression in the `return` statement in the `divide()` method is assignment compatible with the `return` type `double` of the method headers in (d), (e), and (f).

4 Control Flow

4.1 *(d)*

The program will display the letter `b` when run. The second `if` statement is evaluated since the boolean expression of the first `if` statement is `true`. The `else` clause belongs to the second `if` statement. Since the boolean expression of the second `if` statement is `false`, the `if` block is skipped and the `else` clause is executed.

4.2 *(c)*

The `case` label value `2 * iLoc` is a constant expression whose value is `6`, the same as the `switch` expression. Fall-through results in the program output shown in (c).

4.3 *(c)*

(a) contains a `switch` statement. Note that there is no `break` statement associated with the first `case` label, thus execution falls through to the second `case` label and assigns the string `"Composite"` to the reference `result`, which is then printed.
(b) uses a `switch` expression to yield a result. However, it does not provide an exhaustive set of `case` labels and will fail to compile without the `default` label.
(c) uses the identifier `yield` as both a variable name and a contextual keyword in the `yield` statement. There is no fall-through, and the `switch` expression yields the string `"Prime"` which is printed.
(d) is mixing two different types of notations for the `switch` constructs: the arrow notation and the colon notation, which is not permitted.

4.4 *(a)*

The value `1` of the `price` variable matches the `case` constant 1 in the first `case` label, and in this case the discount is calculated by subtracting 1 from the value of `price`, which results in the value of `0`. This code uses a `switch` expression with the arrow notation, so no fall-through to the next `case` label can occur. Case labels do not need to be listed in any particular order. The `switch` expression is exhaustive, because the `case` labels and the `default` label cover the range of `int` values. Code will compile and when executed will yield the value `0`.

4.5 (e)

The loop body is executed twice and the program will print `3`. The first time the loop is executed, the variable `i` changes value from `1` to `2` and the variable `b` changes value from `false` to `true`. Then the loop condition is evaluated. Since `b` is `true`, the loop body is executed again. This time the variable `i` changes value from `2` to `3` and the variable `b` changes value from `true` to `false`. The loop condition is now evaluated again. Since `b` is now `false`, the loop terminates and the current value of `i` is printed.

4.6 (b) and (e)

Both the first and the second numbers printed will be `10`. Both the loop body and the update expression will be executed exactly 10 times. Each execution of the loop body will be directly followed by an execution of the update expression. Afterwards, the condition `j < 10` is evaluated to see whether the loop body should be executed again.

4.7 (f)

The code will compile without error, but will never terminate when run. All the sections in the `for` header are optional and can be omitted (but not the semicolons). An omitted loop condition is interpreted as being `true`. Thus a `for(;;)` loop with an omitted loop condition will never terminate, unless an appropriate control transfer statement is encountered in the loop body. The program will enter an infinite loop at (4).

4.8 (a) and (d)

"`i=1, j=0`" and "`i=2, j=1`" are part of the output. The variable `i` iterates through the values `0`, `1`, and `2` in the outer loop, while `j` toggles between the values `0` and `1` in the inner loop. If the values of `i` and `j` are equal, the printing of the values is skipped and the execution continues with the next iteration of the outer loop. The following can be deduced when the program is run: variables `i` and `j` are both `0` and the execution continues with the update expression of the outer loop. "`i=1, j=0`" is printed and the next iteration of the inner loop starts.

Variables `i` and `j` are both `1` and the execution continues with the update expression of the outer loop. "`i=2, j=0`" is printed and the next iteration of the inner loop starts. "`i=2, j=1`" is printed, `j` is incremented, `j < 2` is `false`, and the inner loop ends. Variable `i` is incremented, `i < 3` is `false`, and the outer loop ends.

4.9 (c) and (d)

The element type of the array `nums` must be assignment compatible with the type of the loop variable (i.e., `int`). Only the element type in (c), `Integer`, can be automatically unboxed to an `int`. The element type in (d) is `int`.

4.10 (d) and (e)

In the header of a `for(:)` loop, we can only declare one local variable. This rules out (a) and (b), as they specify two local variables. Also, the array expression in (a), (b), and (c) is not valid. Only (d) and (e) specify a legal `for(:)` header.

4.11 (a), (b), and (c)

Changing the value of the *`variable`* does not affect the data structure being iterated over. The `for(:)` loop cannot run backwards. We cannot iterate over several data structures simultaneously in a `for(:)` loop, as the syntax does not allow it.

5 Object-Oriented Programming

5.1 (a) and (c)

`Bar` is a subclass of `Foo` that overrides the method `g()`. The statement `a.j = 5` is not legal, since the member `j` in the class `Bar` cannot be accessed through a `Foo` reference. The statement `b.i = 3` is not legal either, since the `private` member `i` cannot be accessed from outside of the class `Foo`.

5.2 (g)

It is not possible to invoke the `doIt()` method in `A` from an instance method in class `C`. The method in `C` needs to call a method in a superclass two levels up in the inheritance hierarchy. The `super.super.doIt()` strategy will not work, since `super` is a keyword and cannot be used as an ordinary reference, nor accessed like a field. If the member to be accessed had been a field or a `static` method, the solution would be to cast the `this` reference to the class of the field and use the resulting reference to access the field, as illustrated in (f). Field access is determined by the declared type of the reference, whereas the instance method to execute is determined by the actual type of the object denoted by the reference at runtime.

5.3 (e)

The code will compile without errors. None of the calls to a `max()` method are ambiguous. When the program is run, the `main()` method will call the `max()` method on the `C` object referred to by the reference `b` with the parameters `13` and `29`. This method will call the `max()` method in `B` with the parameters `23` and `39`. The `max()` method in `B` will in turn call the `max()` method in `A` with the parameters `39` and `23`. The `max()` method in `A` will return `39` to the `max()` method in `B`. The `max()` method in `B` will return `29` to the `max()` method in `C`. The `max()` method in `C` will return `29` to the `main()` method.

5.4 (g)

In the class `Car`, the static method `getModelName()` hides the `static` method of the same name in the superclass `Vehicle`. In the class `Car`, the instance method `getRegNo()` overrides the instance method of the same name in the superclass `Vehicle`. The declared type of the reference determines the method to execute when a `static` method is called, but the actual type of the object at runtime determines the method to execute when an overridden method is called.

5.5 (e)

The class `MySuper` does not have a no-argument constructor. This means that constructors in subclasses must explicitly call the superclass constructor and provide the required parameters. The supplied constructor accomplishes this by calling `super(num)` in its first statement. Additional constructors can accomplish this either by calling the superclass constructor directly using the `super()` call, or by calling another constructor in the same class using the `this()` call which in turn calls the superclass constructor. (a) and (b) are not valid, since they do not call the superclass constructor explicitly. (d) fails, since the `super()` call must always be the first statement in the constructor body. (f) fails, since the `super()` and `this()` calls cannot be combined.

5.6 (b)

In a subclass without any declared constructors, the implicit default constructor will call `super()`. Use of the `super()` and `this()` statements is not mandatory as long as the superclass has a no-argument constructor. If neither `super()` nor `this()` is declared as the first statement in the body of a constructor, then the default `super()` will implicitly be the first statement. A constructor body cannot have both a `super()` and a `this()` statement. Calling `super()` will not always work, since a superclass might not have a no-argument constructor.

5.7 *(d)*

The program will print `12` followed by `Test`. When the `main()` method is executed, it will create a new instance of `B` by passing `"Test"` as an argument. This results in a call to the constructor of `B` that has one `String` parameter. The constructor does not explicitly call any superclass constructor nor any overloaded constructor in `B` using a `this()` call, but instead the no-argument constructor of the superclass `A` is called implicitly. The no-argument constructor of `A` calls the constructor in `A` that has two `String` parameters, passing it the argument list `("1", "2")`. This constructor calls the constructor with one `String` parameter, passing the argument `"12"`. This constructor prints the argument, after implicitly invoking the no-argument constructor of the superclass `Object`. Now the execution of all the constructors in `A` is completed, and execution continues in the constructor of `B`. This constructor now prints the original argument `"Test"` and returns to the `main()` method.

5.8 *(c)*

Any non-`final` class can be declared `abstract`. A class cannot be instantiated if the class is declared `abstract`. The declaration of an `abstract` method cannot provide an implementation. The declaration of a non-`abstract` method must provide an implementation. If any method in a class is declared `abstract`, then the class must be declared `abstract`, so (a) is invalid. The declaration in (b) is not valid, since it omits the keyword `abstract` in the method declaration. The declaration in (d) is not valid, since it omits the keyword `class`. In (e), the return type of the method is missing.

5.9 *(b)*

Since the method is `abstract`, it cannot be inserted at (1) because class `Animal` is not `abstract`—thus ruling out (a) and (c). Class `Cat` is `abstract`, and the method can be inserted at

(2)—thus ruling out (d).

5.10 (d)

We cannot create an object of an `abstract` class with the `new` operator.

5.11 (d)

An instance of `Bacteria` can be assigned to the `org` variable at (2), since a supertype reference can refer to a subtype object. There is no `@Overload` annotation.

5.12 (a) and (b)

The `extends` clause is used to specify that a class extends another class. A subclass can be declared `abstract` regardless of whether the superclass was declared `abstract`. Private, overridden, and hidden members from the superclass are not inherited by the subclass. A class cannot be declared both `abstract` and `final`, since an `abstract` class needs to be extended to be useful, and a `final` class cannot be extended. The accessibility of the class is not limited by the accessibility of its members. A class with all members declared `private` can still be declared `public`.

5.13 (c)

Only a `final` class cannot be extended, as in (c). (d) will fail to compile. A class cannot be declared both `final` and `abstract`, as in (d).

5.14 (c)

Line (3), `void k() { i++; }`, can be re-inserted without introducing errors. Reinserting line (1) will cause the compilation to fail, since `MyOtherClass` will try to override a `final` method. Re-inserting line (2) will fail, since `MyOtherClass` will no longer have the no-argument constructor. The `main()` method needs to call the no-argument constructor. Re-inserting line (3) will work without any problems, but reinserting line (4) will fail, since the method will try to access a `private` member of the superclass.

5.15 (a) and (c)

Abstract classes can declare both `final` methods and non-`abstract` methods. Non-`abstract` classes cannot, however, contain `abstract` methods. Nor can `abstract` classes be `final`. Only interfaces can declare `default` methods.

5.16 (d)

There is no problem compiling the code.

5.17 (a)

A `final` class cannot have `abstract` methods, as a `final` class is a concrete class, providing implementation for all methods in the class.

5.18 (b) and (g)

The keywords `protected` and `final` cannot be applied to interface methods. The keyword `public` is implied, but can be specified for abstract and default interface methods. The keywords `private`, `default`, `abstract`, and `static` can be specified for `private`, `default`, `abstract`, and `static` methods, respectively. The keywords `private`, `default`, and `static` are required for `private`, `default`, and `static` methods, respectively, but the keyword `abstract` is optional as an `abstract` method is understood to be implicitly `abstract`.

5.19 (e)

The `static` method `printSlogan()` is *not* inherited by the class `Firm`. It can only be invoked by using a static reference—that is, the name of the interface in which it is declared, regardless of whether the call is in a static or a non-static context.

5.20 (c)

The instance method at (3) overrides the `default` method at (1). The `static` method at (2) is not inherited by the class `RaceA`. The instance method at (4) does not override the `static` method at (2).

The method to invoke by the call at (5) is determined at runtime by the object type of the reference, which in this case is `Athlete`, resulting in the method at (3) being invoked. Similarly, the call at (6) will invoke the instance method at (4).

5.21 (d)

The code will compile without errors. The class `MyClass` declares that it implements the interfaces `Interface1` and `Interface2`. Since the class is declared `abstract`, it does not need to implement all `abstract` method declarations defined in these interfaces. Any non-`abstract` subclasses of `MyClass` must provide the missing method implementations. The two interfaces share a common `abstract` method declaration, `void g()`. `MyClass` provides an implementation for this `abstract` method declaration that satisfies both `Interface1` and `Interface2`. Both interfaces provide declarations of constants named `VAL_B`. This can lead to ambiguity when referring to `VAL_B` by its simple name from `MyClass`. The ambiguity can be resolved by using the qualified names `Interface1.VAL_B` and `Interface2.VAL_B`. However, there are no problems with the code as it stands.

5.22 *(b)*

The compiler will allow the statement, as the cast is from the supertype (`Super`) to the subtype (`Sub`). However, if at runtime the reference `x` does not denote an object of the type `Sub`, a `ClassCastException` will be thrown.

5.23 *(b)*

The expression `(o instanceof B)` will return `true` if the object referred to by `o` is of type `B` or a subtype of `B`. The expression `(!(o instanceof C))` will return `true` unless the object referred to by `o` is of type `C` or a subtype of `C`. Thus the expression `(o instanceof B) && (!(o instanceof C))` will only return `true` if the object is of type `B` or a subtype of `B` that is not `C` or a subtype of `C`. Given objects of the classes `A`, `B`, and `C`, this expression will only return `true` for objects of class `B`.

5.24 *(d)*

The program will print all the letters `I`, `J`, `C`, and `D` at runtime. The object referred to by the reference `x` is of class `D`. Class `D` extends class `C` and implements `J`, and class `C` implements interface `I`. This makes `I`, `J`, and `C` supertypes of class `D`. The reference value of an object of class `D` can be assigned to any reference of its supertypes and is, therefore, an `instanceof` these types.

5.25 *(c)*

The calls to the `compute()` method in the method declarations at (2) and at (3) are to the `compute()` method declaration at (1), as the argument is always an `int[]`.
The method call at (4) calls the method at (2). The signature of the call at (4) is

```
compute(int[], int[])
```

which matches the signature of the method at (2). No implicit array is created. The method call at (5) calls the method at (1). An implicit array of `int` is created to store the argument values.

The method calls at (6) and (7) call the method at (3). Note the type of the variable arity parameter at (3): an `int[][]`. The signature of the calls at (6) and (7) is

```
compute(int[], int[][])
```

which matches the signature of the method at (3). No implicit array is created.

5.26 (f)

The `instanceof` pattern match operator can introduce a pattern variable in certain boolean expressions. In the conditional of the `if` statement, both operands of the short-circuit `&&` operator must be `true` for the pattern variable `s` to be introduced in the `if` block—the scope of variable `s` is then the `if` block, and `s` is not accessible in the `else` block. The variable `s` is thus out of scope in the `else` block, and the code will not compile.

5.27 (d)

For the `instanceof` pattern match operator, the pattern type (i.e., the type specified for the right operand) must be a subtype of the expression type (i.e., the type of the left operand). This is not the case in (a), (b), or (e). In (a) and (b), both the pattern type and the expression type are `Integer`, and in (e), the pattern type `Number` is a supertype of the expression type `Integer`. Thus (a), (b), and (e) will result in a compile-time error.

In (c), the expression type `Integer` is incompatible with the pattern type `String` for comparing types, as one cannot be cast to the other, thus resulting in a compile-time error.

In (d), the pattern type `Integer` is a subtype of the expression type `Number` and will compile without any problem.

5.28 (a) and (c)

An `instanceof` pattern match operator returns `false` if the reference is `null`; therefore, it will not throw a `NullPointerException`. A pattern variable is only introduced when the `instanceof` pattern match operator returns `true`.

5.29 (e)

The program will print `2` when `System.out.println(ref2.f())` is executed. The object referenced by `ref2` is of class `C`, but the reference is of type `B`. Since `B` contains a method `f()`, the method call will be allowed at compile time. During execution, it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method in `C` to be executed.

5.30 *(c)*

The program will print `1` when run. The `f()` methods in `A` and `B` are `private`, and are not accessible by the subclasses. Because of this, the subclasses cannot overload or override these methods, but simply define new methods with the same signature. The object being called is of class `C`. The reference used to access the object is of type `B`. Since `B` contains a method `g()`, the method call will be allowed at compile time. During execution, it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method `g()` in `B` to be executed. This method calls a method named `f`. It can be determined during compilation that this can only refer to the `f()` method in `B`, since the method is `private` and cannot be overridden. This method returns the value `1`, which is printed.

5.31 *(b), (c), and (d)*

The code as it stands will compile. The use of inheritance in this code defines a `Planet` *is-a* `Star` relationship. The code will fail if the name of the field `starName` is changed in the `Star` class, since the subclass `Planet` tries to access it using the name `starName`. An instance of `Planet` is not an instance of `HeavenlyBody`. Neither `Planet` nor `Star` implements `HeavenlyBody`.

5.32 *(d)*

An enum type can be run as a standalone application, if it provides the appropriate `main()` method. The constants need not be qualified when referenced inside the enum type declaration. The constants *are* `static` members. The `toString()` method always returns the name of the constant, unless it is overridden.

5.33 *(b)*

(1), (2), and (3) define *constant-specific class bodies* that override the `toString()` method. For constants that do not override the `toString()` method, the name of the constant is returned.

5.34 *(c)*

An enum type cannot be instantiated to create more objects than those already created implicitly for its constants. All enum types override the `equals()` method from the `Object` class. The `equals()` method of an enum type compares its constants for equality according to reference equality (the same as with the `==` operator) based on their ordinal values. This `equals()` method is `final`.

5.35 *(d) and (e)*

In (a), the compiler recognizes a non-canonical constructor with no parameters in the record class definition. The first statement in such a non-canonical constructor must be an explicit invocation of the canonical constructor using the `this()` expression. For example, the following constructor declaration will compile, but it will not give the desired result.

```
public Product() {
  this(0, "No name", 0.00);
}
```

However, specifying the required parameters in the constructor header will result in the normal canonical constructor that will compile, and the code will print the right result:

Click here to view code image

```
public Product(int id, String name, double price) {
...
}
```

(b) does not compile because the parameter names in the normal canonical constructor do not match the ones defined in the header of the record class.

In (c), the compiler recognizes a record class that has no component fields. The constructor declared is a non-canonical constructor that must have an explicit invocation of the no-argument implicit canonical constructor using the `this()` expression. For example, the following constructor declaration will compile, but will not give the desired result.

Click here to view code image

```
public Product(int id, String name, double price) {
  this();
}
```

However, specifying the field components in the record class header will make the code compile and give the right result:

Click here to view code image

```
public record Product(int id, String name, double price) {
...
}
```

(d) correctly initializes a record using the compact constructor. The name will be stored in uppercase.

(e) correctly initializes a record using the implicit canonical constructor. The record class overrides the method `toString()` that returns the `name` field value represented as an uppercase `String`.

(f) correctly initializes a record using the implicit canonical record constructor, but its overridden `toString()` method accesses the fields directly, without converting the `name` field to uppercase. It does not invoke the `name()` method.

5.36 *(d)*

The compiler automatically generates an implementation of the `equals()` method for a record class, if one is not provided. The `equals()` method added by the compiler will compare all component fields of the record class. This means that the `equals()` method will return `true`, but the equality operator `==` will return `false`, as the two records that are created are distinct objects that have the same state.

5.37 *(c)*

All component fields defined by a record class are immutable. A record class can only declare static fields in addition to the component fields specified in its header. The compiler automatically generates the get methods for the component fields of the record class, but not the set methods, since such fields are immutable. Record classes implicitly extend the `java.lang.Record` class. Record classes cannot have an explicit `extends` clause.

5.38 *(c)*

Sealed classes can be `abstract`. In fact, this is often the case, as the `abstract sealed` class is intended to be extended by its permitted subclasses. A `non-sealed` class can also be `abstract` and can be freely extended. However, a `sealed` class can only be extended by its permitted subclasses. A class that extends a `sealed` class must be either `final`, `sealed`, or `non-sealed`.

5.39 *(c)*

In the code, subtypes `Y` and `Z` can be interfaces or classes that can either extend or implement the `sealed` interface `X`. A class or an interface that is marked `sealed` must be defined with the `permits` clause that specifies its permitted subtypes, unless the permitted subtypes are specified in the same compilation unit. Since the classes and interfaces are all `public`, each is defined in its own compilation unit.

In (a), interface `Z` is marked `sealed`, but does not provide the `permits` clause or its permitted subtypes in the same compilation unit.

A permitted subtype of a `sealed` supertype must be explicitly marked as either `final`, `non-sealed`, or `sealed`. In (b), interface `Z` is not marked with any of these markers, so it will not compile. (d) has the exact same problem with class `Y`.

In (c), interface `Z` is correctly marked as `sealed`, with the appropriate `permits` clause, and class `Y` correctly implements both its `sealed` superinterfaces `X` and `Z`.

6 Access Control

6.1 (a) and (c)

Bytecode of all reference type declarations in the file is placed in the designated package, and all reference type declarations in the file can access the imported types.

6.2 (e)

Both classes are in the same package `app`, so the first `2` import statements are unnecessary. The package `java.lang` is always imported in all compilation units, so the next two import statements are unnecessary. The last static import statement is necessary to access the static variable `frame` in the `Window` class by its simple name.

6.3 (b), (c), (d), and (e)

In (a), the import statement imports types from the `mainpkg` package, but `Window` is not one of them.
In (b), the import statement imports types from the `mainpkg.subpkg1` package, and `Window` is one of them.
In (c), the import statement imports types from the `mainpkg.subpkg2` package, and `Window` is one of them.
In (d), the first import statement is a type-import-on-demand statement and the second import statement is a single-type-

import statement. Both import the type `Window`. The second one overrides the first one.
In (e), the first import statement is a single-type-import statement and the second import statement is a type-import-on-demand statement. Both import the type `Window`. The first one overrides the second one.
In (f), both import statements import the type `Window`, making the import ambiguous.
In (g), both single-type-import statements import the type `Window`. The second import statement causes a conflict with the first one.

6.4 *(c) and (e)*
The name of the class must be fully qualified. A parameter list after the method name is not permitted. (c) illustrates single static import, and (e) illustrates static import on demand.

6.5 *(b) and (d)*
In (a) and (c), class `A` cannot be found. In (e) and (f), class `B` cannot be found—there is no package under the current directory `/top/wrk/pkg` to search for class `B`. Note that specifying `pkg` in the classpath in (d) is superfluous. The *parent* directory of the package must be specified—that is, the *location* of the package.

6.6 *(d) and (e)*
Static field `y` in class `a.b.X` is accessed in the method `xyz()` of class `a.b.c.Z`. Static import allows static members from reference type declarations in other packages to be accessed by their simple names.
This rules out (a) as it is a type-import-on-demand statement for all reference type declarations in package `a.b`, and also (b) as it is a type-import-on-demand statement from class `a.b.X`. (a) imports class `X`, but (b) does not import any type, as class `X` does not declare any non-static inner class members.
(d) is a static import-on-demand statement, meaning it imports all static members of the class `a.b.X`, including `y` which can be accessed by its simple name. (e) is a single-static-import statement, meaning only the designated static member `y` from class `a.b.X` is imported and can be accessed by its simple name.

6.7 *(a) and (d)*
The class `Farm` in package `habitat` accesses classes `Cat` and `Cow` by their simple names from package `life.animals`. (a) is a

type-import-on-demand of all reference type declarations from package `life.animals`, including `Cat` and `Cow`. (b) and (c) are ruled out as these are static imports. (d) imports the classes `Cat` and `Cow` individually.

6.8 (d)

Packages are typically mapped to directories in a file system. A subpackage is an autonomous package that just happens to map to a subdirectory of a directory that represents some other package. There is no relationship between a package and its subpackages. Each package is treated independently, regardless of whether it appears to be implemented as a subdirectory, ruling out (a) and (b). (c) is incorrect because reference types and static members of types in other packages can be accessed by their fully qualified names, rather than using import statements.

Import statements are not present in the compiled code at all, as type names are always replaced with fully qualified names by the compiler.

6.9 (b) and (e)

If no access modifier (`public`, `protected`, or `private`) is given in the member declaration of a class, the member is only accessible by classes in the same package.

A subclass does not have access to members with package accessibility declared in a superclass, unless they are in the same package.

Local variables cannot be declared `static` or have an access modifier.

6.10 (b)

Outside the package, the member `j` is accessible to any class, whereas the member `k` is only accessible to subclasses of `MyClass`.

The field `i` has package access, and is only accessible by classes inside the package. The field `j` has public access, and is accessible from anywhere. The field `k` has protected access, and is accessible from any class inside the package and from subclasses anywhere. The field `l` has private access, and is only accessible within the class itself.

6.11 (b)

A private member is only accessible in the class in which it is declared. If no access modifier has been specified for a member, the member has package accessibility. The keyword `default` is not an access modifier. A member with package access is only accessible from classes in the same package. Subclasses in other packages cannot access a member with package accessibility.

6.12 *(d)*

A class that is declared as `final` cannot be extended. Making a class `final` is not enough to prevent its state from being modified. A static modifier can be applied to inner classes, but this is not relevant to the question of immutability. A field within an immutable object can refer to a mutable object, which means that members of an immutable object are not automatically immutable.

6.13 *(a)*

In (a), marking the field `name` `private` means it can only be accessible in the class. It can only be initialized once by the constructor when the object is created, and removing the `setName()` method means the value of `private` field `name` cannot be changed. The state of the object is thus immutable. In (b), the assignment in the `setName()` method will not compile as it changes the value of the `final` field `name` which has already been initialized in the constructor. In (c), the assignment in the constructor will not compile as it changes the value of the `final` field `name` which has already been initialized in its declaration.

7 Exception Handling

7.1 *(d)*

The program will only print `1`, `4`, and `5`, in that order. The expression `5/k` will throw an `ArithmeticException`, since `k` equals 0. Control is transferred to the first `catch` clause, since it is the first `catch` clause that can handle the arithmetic exceptions. This exception handler simply prints `1`. The exception has now been caught and normal execution can resume. Before leaving the `try` statement, the `finally` clause is executed. This `finally` clause prints `4`. The last statement of the `main()` method prints `5`.

7.2 *(b) and (e)*

If run with no program arguments, the program will print `The` `end`. If run with one program argument, the program will print the specified argument followed by `The end`. The `finally` clause will always be executed, no matter how control leaves the `try` block.

7.3 (c) and (d)

Normal execution will only resume if the exception is caught by the method. The uncaught exception will propagate up the JVM stack until some method handles it. An overriding method need only declare that it can throw a subset of the checked exceptions the overridden method can throw. The `main()` method can declare that it throws checked exceptions just like any other method. The `finally` clause will always be executed, no matter how control leaves the `try` block.

7.4 (b)

The only thing that is wrong with the code is the ordering of the `catch` and `finally` clauses. If present, the `finally` clause must always appear last in a `try-catch-finally` construct. Note that since `B` is a subclass of `A`, catching `A` is sufficient to catch exceptions of type `B`.

7.5 (b)

An invocation of the `average()` method throws an `ArithmeticException`, which is then caught in the `main()` method. The catch block prints `"error"`. This means that the execution of the `average()` method is stopped, and the method does not return any value, leaving the local variable `value` still initialized to `1`, which is printed.

7.6 (e)

A `null` value is passed as an argument to the `reaction()` method, resulting in a `PlayerException` being thrown, containing the `"Invalid action"` message. This exception is then caught in the `main()` method, where its error message is assigned to the local variable `message` in the `catch` block. As this exception was successfully handled, normal execution resumes. The print statement prints the error message `"Invalid action"`.

7.7 (c)

As a `null` value is passed to the `readFile()` method, it throws a `FileNotFound-Exception`, which is a subclass of `IOException`. This exception is caught by the corresponding `catch` block in the `main()` method, printing `"IO error: invalid file name"`.

Upon resumption of normal execution, the `finally` block prints `" finally"`, followed by the last print statement printing `" the end"`.

7.8 *(g)*

The `readFile()` method executes normally, which means that no `catch` block is executed in the `main()` method. The `finally` block prints `"finally"` and the last print statement prints `" the end"`.

7.9 *(d)*

A `null` value is passed to the `readFile()` method which then throws an unchecked `NullPointerException`, which is a subclass of `RuntimeException`. It is not required to explicitly specify unchecked exceptions in the `throws` clause or to handle them. The `NullPointerException` is propagated to the invoking method `main()`, where it is caught by the `catch` block that catches an `Exception`, since `RuntimeException` is a subclass of `Exception`. The `catch` block prints `"Other error: invalid file name"`. Although this `catch` block contains a `return` statement, the `finally` block is executed first, printing `" finally"`, before returning from the `main()` method. Thus the last print statement in the `main()` method is not executed.

7.10 *(b), (c), and (e)*

(b), (c), and (e) correspond to (2), (3), and (5). `FileNotFoundException` is thrown by the constructor call `FileReader(filename)`. The `close()` method of the `BufferedReader` throws an `IOException`. Either the `try`-with-resources statement must catch it or the exception must be specified in the `throws` clause of the method—the catch-or-declare rule. (1), (4), and (6) do not fulfill this criteria. Also, the resource variables are `final` and cannot be assigned to in the body of the `try`-with-resources statement, ruling out (7). At (5), the resource declaration statements are valid.

7.11 *(h)*

The top-level `try` block in the method `justDoIt()` throws an `IOException`. The nested `try` block in the `finally` clause throws an `EOFException` that is caught and associated as a suppressed exception with the `IOException`. It is the `IOException` that is propagated. The `IOException` is caught in the `catch` clause in the `main()` method and its information is printed, including its suppressed exception `EOFException`. The supertype exception references are used polymorphically to handle objects of subtype exceptions.

7.12 *(f)*

In (a), the program does not compile because the checked `Exception` thrown in the `close()` method does not comply with the catch-or-declare rule.

In (b), although the `close()` method will abide by the catch-or-declare rule, the `main()` method does not.

In (c), adding `throws Exception` clause only to the `main()` method does not change the fact that the `close()` method does not abide by the catch-or-declare rule.

In (d), both methods abide by the catch-or-declare rule. When run, the program will throw an `Exception` that is not caught.

In (e), adding `catch (Exception e) {}` clause to the `try` statement in the `main()` method does not change the fact that the `close()` method does not abide by the catch-or-declare rule.

In (f), the `close()` method will abide by the catch-or-declare rule, and the `main()` method will catch and handle the exception thrown at runtime.

7.13 *(a), (b), and (c)*

In (a), the exception parameter `e` is implicitly `final` and cannot be reassigned in the multi-`catch` clause.

In (b), in the two assignments to the exception parameter `e`, objects of the superclass `IOException` cannot be assigned to references of subtypes `EOFException` and `FileNotFoundException`.

In (c), in the assignment to the exception parameter `e` of type `Exception`, an object of the subtype `IOException` is assigned to `e`, but an exception of type `Exception` is thrown in the `catch` clause. This exception is not covered by the subtype `IOException` specified in the `throws` clause. In other words, `Exception` thrown in the `catch` clause is not handled.

In (d), the compiler can infer that only `FileNotFoundException` can be thrown in the `try` statement. Such an exception can only be thrown in the `catch` clause, as the parameter `e` of type `Exception` can be inferred to be effectively `final`, and can thus only refer to a `FileNotFoundException`. This exception is covered by the `throws` clause.

In (e), the compiler can infer that only `FileNotFoundException` can be thrown in the `try` statement. This exception is caught by parameter `e` of the superclass `IOException`. `IOException` is covered by the `throws` clause that specifies its supertype

`Exception`.

7.14 *(a)*

In this code example, the `Resource` object is used in a `try`-with-resources statement. Its `action()` method will print `"action "` and it will be closed by the implicit `finally` block by invoking the `close()` method that prints `"closure "`. There are no exceptions thrown. The last print statement prints `" the end"`.

7.15 *(b)*

The `Resource` object is used in the `try`-with-resources statement, which means it will be closed by the implicit `finally` block invoking the `close()` method after the execution of the `try` block.
There are two exceptions thrown in the code: The first is an `IOException` that is thrown by the `action()` method, and the second is thrown by the `close()` method of the `Resource` class. The `IOException` is then caught in the `main()` method. However, notice that the `IOException` handler does not attempt to retrieve and print information about suppressed exceptions thrown by the implicit `finally` block of the `try`-with-resources statement. The `catch` block prints `"IO action error "`. Once the exception is handled, execution of the rest of the method `main()` resumes. The last print statement prints `" the end"`.

7.16 *(b)*

There is no reason why explicit and implicit `finally` blocks cannot coexist. If an explicit `finally` block is added after the `try`-with-resources statement, its code is executed after the implicit `finally` block.

8 Selected API Classes

8.1 *(e)*

Neither the `hashCode()` method nor the `equals()` method is declared `final` in the `Object` class, and it cannot be guaranteed that implementations of these methods will differentiate between *all* objects. All arrays are genuine objects and inherit from the `Object` class, including the `clone()` method.

8.2 *(b)*

Values in the range –128 to +127, inclusive, are boxed in `Integer` objects and cached by the method `Integer.valueOf()`.

8.3 *(c)*

There is a minor performance penalty associated with the conversion of a primitive value to a wrapper object and vice versa. Wrapper references can be assigned the `null` value, but they cannot be assigned to a variable of a primitive type. An attempt to convert an uninitialized wrapper reference to a primitive value will result in a `NullPointerException`. However, if the reference is a local variable then the code will not compile.

8.4 *(b)*

`Integer` objects with a value between –128 and +127 are interned. Therefore, two references that reference the same interned `Integer` object will return `true` when compared with the `==` operator—that is, they are aliases. The reference `i1` is assigned the reference value of a new `Integer` object with value `10`. This `Integer` object is interned. The reference `i2` is assigned the reference value of this interned `Integer` object, instead of creating a new `Integer` object. The expression `i1 == i2` is thus `true`, resulting in `A` being printed. The expression `i1 == i3` is also `true`, since the `Integer` object referenced by `i1` is unboxed to the `int` value `10` which is also the value in `i3`, resulting in `B` being printed.
However, values boxed by the references `x1` and `x2` are greater than 127, and therefore these references refer to two different `Integer` objects which are not interned. The expression `x1 == x2` returns the value `false`. The expression `x1 == x3` returns `true`, since the `Integer` object referenced by `x1` is unboxed to the `int` value `1000` which is also the value in `x3`, resulting in `D` being printed.

8.5 *(d)*

The expression `str.substring(2,5)` will extract the substring `"kap"`. The method extracts the characters from index 2 to index 4, inclusive.

8.6 *(d)*

The program will print `str3str1` when run. The `concat()` method will create and return a new `String` object, which is the concatenation of the current `String` object and the `String` object passed as an argument. The expression statement `str1.concat(str2)` creates a new `String` object, but its reference value is not stored after the expression is evaluated. Therefore, this `String` object gets discarded.

8.7 *(d)*

The constant expressions `"ab" + "12"` and `"ab" + 12` will, at compile time, be evaluated to the string-valued constant `"ab12"`. Both variables `s` and `t` are assigned a reference to the same interned `String` object containing `"ab12"`. The variable `u` is assigned a new `String` object, created by using the `new` operator.

8.8 *(b)*

The reference value in the reference `str1` never changes and it refers to the string literal `"lower"` all the time. The calls to `toUpperCase()` and `replace()` return a new `String` object whose reference value is ignored.

8.9 *(d)*

The call to the `putO()` method does not change the `String` object referred to by the `s1` reference in the `main()` method. The reference value returned by the call to the `concat()` method is ignored.

8.10 *(b)*

The reference value in the reference `str1` never changes and it refers to the string literal `"lower"` all the time. The calls to `toUpperCase()` and `replace()` return a new `String` object whose reference value is ignored.

8.11 *(b)*

The `substring()` method returns the characters from the start index inclusive to the end index exclusive. The start index is returned by the `indexOf(' ')` method call, which is the first occurrence of a space character `' '` within the string, namely index 4. The expression `s.indexOf(' ', s.indexOf(' ') + 1)` finds the next occurrence of the space character `' '`, where the search starts after the first occurrence of the space character (`' '`), returning the index 7. As 1 is added to this index, the end index passed to the `substring()` method is 8. The resulting substring is from index 4 inclusive to index 8 exclusive—that is, `" is "`. The `strip()` method removes both leading and trailing whitespace from this string, resulting in the string `"is"`. To this string, the character `'-'` is concatenated at either end.

8.12 (a)

This text block does not have any incidental whitespace because the last line has no leading whitespace before the closing delimiter of the text block. The `while` loop splits the text block into individual lines, extracting a substring from the start to the line terminator (`\n`) of each line. The length of each line does not include the line terminator. The lengths are 3, 5, and 3, as no incidental whitespace is removed. The length of each line is then printed.

8.13 (d)

In (a) and (b), the content of the text block does not start after the line terminator of the opening delimiter (`"""`).

In (c), the text block does not end with the closing delimiter (`"""`), but with four double quotes. Note that there is no requirement that double quotes should be balanced in a text block, and can be specified with or without escaping.

In (d), the text block ends correctly, as it uses the `\"` escape character for the double quote that should be part of the text block, allowing it to be distinguished from the closing delimiter. However, the last line of the block will not end with a line terminator. The resulting string literal is `"\"a\"\"b\""`. When printed, the output will be a single line containing the characters `"a""b"`.

(e) is syntactically correct because the text block is correctly terminated. However, in this case the closing delimiter is on a line on its own, resulting in the last line of the text block content to end with a line terminator. The resulting string literal is `"\"a\"\"b\"\n"`. When printed, the output will be a line containing the characters `"a""b"` followed by a newline.

(f) is incorrect because the last `\"` escape character results in the subsequent two double quotes also to be escaped, resulting in no closing delimiter being found— that is, `\"""` results in `\"\"\"`.

8.14 (a) and (e)

The content of a text block starts on a new line of text immediately after the line that contains the opening delimiter, and ends just before the closing delimiter. This makes (a) correct, but not (b).

A text block is not a subtype of the `String` class, as the `String` class is final, and the type of a text block is `String`.

Although trailing whitespace is removed from the end of each line in the text block, only incidental whitespace is removed from the start of each line in the text block.

8.15 *(a)*

The code will fail to compile, since the expression `(s == sb)` is illegal. It compares references of two classes that are not related. Also, the `StringBuffer` class does not override the `equals()` method from the `Object` class, but inherits it.

8.16 *(e)*

The program will compile without errors and will print `Have a` when run. The contents of the string buffer are truncated to six characters by the method call `sb.setLength(6)`.

8.17 *(c)*

The `trimtoSize()` only changes the capacity to match the length of the string builder. It does not the change the length of the string builder. The methods `append()`, `reverse()`, and `setLength()` change the string builder successively by appending `"!"` (`" 1234 !"`), reversing the string builder (`"! 4321 "`), and setting the length to 5 (`"! 43"`). The print statement prints `|! 43|`.

8.18 *(b)*

The references `sb1` and `sb2` are not aliases. The `StringBuilder` class does not override the `equals()` method so the result will be the same as with the `==` operator. The correct answer is (b).

8.19 *(a)*

The `StringBuilder` class does not override the `hashCode()` method, but the `String` class does. The references `s1` and `s2` refer to a `String` object and a `StringBuilder` object, respectively. The hash values of these objects are computed by the `hashCode()` method in the `String` and the `Object` class, respectively—giving different results. The references `s1` and `s3` refer to two different `String` objects that are equal, hence they have the same hash value.

8.20 *(b)*

String builders are mutable. When created, the string builder `s1` has the sequence `"W"`. The call to the `append()` method in the `putO()` method appends `"O"`, resulting in `"WO"`. On return from the `put0()` method, the call to the `append()` method in the `main()` method appends `"W!"` to the string builder. The string builder `s1` now contains the sequence `"WOW!"` which is printed.

8.21 (i)

A `StringBuilder` is manipulated by different methods. First, the string `"12"` is appended, then the string `"34"` is inserted at index 1, resulting in the string `"1342"` in the `StringBuilder` object. Next, the `delete()` method does not modify the contents because the start and the end indexes are the same. Finally, the `replace()` method replaces the characters between the start indices 0 inclusive and the end index 1 exclusive with an empty string—that is, effectively removing the character `'1'` from index 0. The resulting string is `"342"`.

8.22 (b)

Remember that the default capacity of the empty `StringBuilder` is 16 characters, which can change as its contents are modified. The string `"42"` is appended first, then the second character is deleted from this string, resulting in the `StringBuilder` object containing the string `"4"`. The print statement concatenates the string `"4"` in the `StringBuilder` with the sum of its capacity (which still has the default value 16) and its length (which is 1)—in other words, the string `"4"` is concatenated with `17`. The resulting string `"417"` is printed.

8.23 (b) and (d)

The method call `Math.ceil(v)` returns the `double` value `11.0`, which is printed as `11.0` at (1), but as `11` at (4) after conversion to an `int`.
The method call `Math.round(v)` returns the `long` value `11`, which is printed as `11` at (2).
The method call `Math.floor(v)` returns the `double` value `10.0`, which is printed as `10.0` at (3), but as `10` at (5) after conversion to an `int`. (b) and (d), corresponding to (2) and (4), will print `11`.

8.24 (b)

The value `-0.5` is rounded up to `0` and the value `0.5` is rounded up to `1`.

8.25 (b), (c), and (d)

The expression will evaluate to one of the numbers 0, 1, 2, or 3. Each number has an equal probability of being returned by the expression.

9 Nested Type Declarations

9.1 (e)

The code will compile and print `123` at runtime. An instance of the `Outer` class will be created and the field `secret` will be initialized to `123`. A call to the `createInner()` method will return the reference value of the newly created `Inner` instance. This object is an instance of a non-static member class and is associated with the outer instance. This means that an object of a non-static member class has access to the members within the outer instance. Since the `Inner` class is nested in the class containing the field `secret`, this field is accessible to the `Inner` instance, even though the field `secret` is declared `private`.

9.2 (b) and (e)

A static member class is in many respects like a top-level class, and can contain non-`static` fields. Instances of non-static member classes are created in the context of an outer instance. The inner instance is associated with the outer instance. Several non-static member class instances can be created and associated with the same outer instance. Static member classes do not have any implicit outer instance. A static member interface, just like top-level interfaces, cannot contain non-`static` fields. Nested interfaces are always `static`.

9.3 (d)

The program will compile without error, and will print `1`, `3`, `4`, in that order, at runtime. The expression `B.this.val` will access the value `1` stored in the field `val` of the (outer) `B` instance associated with the (inner) `C` object referenced by the reference `obj`. The expression `C.this.val` will access the value `3` stored in the field `val` of the `C` object referenced by the reference `obj`. The expression `super.val` will access the field `val` from `A`, the superclass of `C`.

9.4 (c) and (d)

The class `Inner` is a non-static member class of the `Outer` class, and its qualified name is `Outer.Inner`. The `Inner` class does not inherit from the `Outer` class. The method named `doIt` is, therefore, neither overridden nor overloaded. Within the scope of the `Inner` class, the `doIt()` method of the `Outer` class is hidden by the `doIt()` method of the `Inner` class.

9.5 (e)

Non-static member classes, unlike top-level classes, can have any access modifier. Static member classes can be declared in a top-level class and any nested class. Methods in all nested classes can be declared `static`. Only static member classes can be declared `static`. Declaring a class `static` only means that instances of the class are created without having an outer instance. This has no bearing on whether the members of the class can be `static` or not.

9.6 (c), (d), and (e)

The method at (1) will not compile, since the parameter `i` is neither `final` nor effectively `final`, and therefore not accessible from within the inner class. The syntax of the anonymous class in the method at (2) is not correct, as the empty argument list is missing. The parameter `i` at (3) is effectively `final`, and at (4) it is `final`. The method at (5) is legally declared.

9.7 (d)

Other static members, not only `static final` fields declared as constant variables, can be declared within a non-static member class. Members in outer instances are directly accessible using simple names (provided they are not hidden). Fields in nested static member classes need not be `final`. Anonymous classes cannot have constructors, since they have no names. Nested classes define types that are distinct from the enclosing class, and the `instanceof` type comparison operator does not take the type of the outer instance into consideration.

9.8 (d)

Note that the nested classes are locally declared in a static context. (a) and (b) refer to the field `str1` in `Inner`. (c) refers to the field `str1` in `Access`. (e) requires the `Helper` class to be in the `Inner` class in order to compile, but this will not print the right answer. (f), (g), and (h) will not compile, as the `Helper` local class cannot be accessed using the enclosing class name.

9.9 (c)

The field `t` denotes an instance of the anonymous inner class that extends the `Test` class. The `toString()` method is implicitly called on `t` in the print statement. The anonymous inner class overrides the `toString()` method, which is invoked. It returns the result of the following `return` statement:

Click here to view code image

```
return this.x + super.toString() + x;
```

Here, both `this.x` and `x` refer to the field `x` declared in the anonymous class, which has the character value `'>'`. This field shadows the local variable `x` in the `main()` method, which in turn shadows the field `x` in the `Test` class.
The call `super.toString()` results in the `toString()` method in the superclass `Test` to be invoked. It returns the result of the following statement:

```
return x + "42";
```

Here, the `x` refers to the field `x` in the `Test` class, which has the character value `'='`. The statement returns the string `"=42"`.
The print statement concatenates the following expression to print `">=42>"`—that is, (c):

```
'>' + "=42" + '>'
```

9.10 *(d)*

The `String` class is `final`, and therefore, cannot be extended. An anonymous inner class tries to extend the `String` class, but it will be flagged as an error by the compiler.

10 Object Lifetime

10.1 *(d)*

An object is only eligible for garbage collection if all remaining references to the object are from other objects that are also eligible for garbage collection. Therefore, if object `obj2` is eligible for garbage collection and object `obj1` contains a reference to it, then object `obj1` must also be eligible for garbage collection. Java does not have a keyword `delete`. An object will not necessarily be garbage collected immediately after it becomes unreachable. However, the object will be eligible for garbage collection. Circular references do not prevent objects from being garbage collected, only reachable references do. An object is not eligible for garbage collection as long as the object can be accessed by any live thread.

10.2 *(b)*

Before (1), the `String` object initially referenced by `arg1` is denoted by both `msg` and `arg1`. After (1), the `String` object is only denoted by `msg`. At (2), the reference `msg` is assigned a new reference value. This reference value denotes a new `String` object created by concatenating the contents of several other `String` objects. After (2), there are no references to the `String` object initially referenced by `arg1`. The `String` object is now eligible for garbage collection.

10.3 *(a)*

The only object created is the array, and it is reachable when control reaches (1).

10.4 *(a)*

All the objects created in the loop are reachable via `p`, when control reaches (1).

10.5 *(a)*

It may seem that since the method `removeAll()` sets the `songs` array reference to `null`, there would be three objects (i.e., the array itself and its two `Song` objects) eligible for garbage collection when control reaches (1). However, prior to this method invocation, this array reference is also assigned to a local array variable `songs` declared in the `main()` method. As a result, even though the `songs` array field in the `Album` object no longer references the `Song` array, the local array variable `songs` still references this array object, which is thus reachable.

10.6 *(c), (e), and (f)*

The `static` initializer blocks (a) and (b) are not legal, since the fields `alive` and `STEP` are non-`static` and `final`, respectively. (d) is not a syntactically legal `static` initializer block. The `static` block in (e) will have no effect, as its body is an empty block. The `static` block in (f) will change the value of the `static` field `count` from 5 to 1.

10.7 *(c)*

The program will compile and print `50, 70, 0, 20, 0` at runtime. All fields are given default values unless they are explicitly initialized. Field `i` is assigned the value `50` in the `static` initializer block that is executed when the class is initialized. This assignment will override the explicit initialization of field `i` in its declaration statement. When the `main()` method is executed, the `static` field `i` is `50` and the `static` field `n` is `0`. When an instance of the class is created using the `new` operator, the value of the `static` field `n` (i.e., `0`) is passed to the constructor. Before the body of the constructor is executed, the instance initializer block is executed, which assigns the values `70` and `20` to the fields `j` and `n`, respectively. When the body of the constructor is executed, the fields `i`, `j`, `k`, and `n`, and the parameter `m`, have the values `50`, `70`, `0`, `20`, and `0`, respectively.

10.8 *(f)*

This class has a blank `final boolean` instance variable `active`. This variable must be initialized when an instance is constructed, or else the code will not compile. This also applies to blank `final static` variables. The keyword `static` is used to signify that a block is a `static` initializer block. No keyword is used to signify that a block is an instance initializer block. (a) and (b) are not instance initializer blocks, and (c), (d), and (e) fail to initialize the blank `final` variable `active`.

10.9 *(c)*

The program will compile and print `2`, `3`, and `1` at runtime. When the object is created and initialized, the instance initializer block is executed first as it is declared first, printing `2`. Then the instance initializer expression is executed, printing `3`. Finally, the constructor body is executed, printing `1`. The forward reference in the instance initializer block is legal, as the use of the field `m` is on the left-hand side of the assignment.

10.10 *(c)*

This question tests understanding of execution order of initializers and constructors when an object is created. First the static initializers are executed, when classes `Music` and `Song` are loaded into memory. Therefore, the string `"-C--F-"` is printed first. The static initializers are invoked only once, so neither `"-C-"` nor `"-F-"` is printed again. This excludes (a) and (b).
When the first new `Song()` object is created, it first triggers initialization starting from its superclass instance initializer and constructor, which prints `"-D--E-"`, after which the instance initializer and constructor in the `Song` class are executed, printing `"-G--A-"`. This process is repeated for the second new song, resulting in `"-D--E--G--A-"` being printed. The final printout is `"-C--F--D--E--G--A--D--E--G--A-"`.

10.11 *(c) and (e)*

Line (1) will cause illegal redefinition of the field `width`. Line

(2) uses an illegal forward reference to the fields `width` and `height`. The assignment in line (3) is legal. Line (4) is an assignment statement, and therefore illegal in this context. Line (5) declares a local variable inside an initializer block with the same name as the instance variable `width`, which is allowed. The simple name in this block will refer to the local variable. To access the instance variable `width`, the `this` reference must be used in this block.

11 Generics

11.1 *(b)*

The type of `intList` is `List` of `Integer` and the type of `numList` is `List` of `Number`. The compiler issues an error because `List<Integer>` is *not* a subtype of `List<Number>`.

11.2 *(c)*

With a reference of type `List<? super Integer>`, a set/put/write/add operation can only add an `Integer` or a subtype of `Integer` to the list. Calls to the `add()` method in the code are not a problem, as an `Integer` is added to the list.
With a reference of type `List<? super Integer>`, a get/read operation can only get an `Object` from the list. This object is not assignable to a reference of type `Number`. (3) will not compile.

11.3 *(b)*

The compiler issues an unchecked conversion warning at (1), as we are assigning a raw list to a generic list.

11.4 *(b), (f), and (g)*

We cannot create an array of a type parameter, as at (2). We cannot refer to the type parameters of a generic class in a static context—for example, in static initializer blocks, static field declarations, and as types of local variables in static methods, as at (6) and (7).

11.5 *(b), (c), (e), and (f)*

In (b), (c), (e), and (f), the parameterized type in the object creation expression is a subtype of the type of the reference. This is not the case in (a): Just because `HashMap<Integer, String>` is a subtype of `Map<Integer, String>`, it does not follow that `HashMap<Integer, HashMap<Integer, String>>` is a subtype of `Map<Integer, Map<Integer, String>>`—there is no subtype covariance relationship between concrete parameterized types. In (d) and (g), wild cards cannot be used to instantiate the class.

11.6 (b)

`ArrayList<Fruit>` is not a subtype of `List<? extends Apple>` at (1), and `ArrayList<Apple>` is not a subtype of `List<? super Fruit>` at (4). Any generic list can be assigned to a raw list reference. A raw list and an unbounded wildcard list are assignment compatible.

11.7 (d)

The compiler issues unchecked warnings for calls to the `add()` method. The `TreeSet` class orders elements according to their natural ordering. A `ClassCastException` is thrown at runtime when the statement `set.add(2)` is executed, as an `Integer` is not comparable to a `String`.

11.8 (a) and (b)

The type of reference `g` is of raw type `Garage`. We can put any object in such a `Garage`, but only get `Object`s out. The type of value returned by the `get()` method at (6) through (8) is `Object`, and therefore, is not assignment compatible with `Vehicle`, `Car`, or `Sedan`.

11.9 (d), (e), and (f)

In (a), the arguments in the call are `(List<Number>, List<Integer>)`. No type inferred from the arguments satisfies the formal parameters `(List<? extends T>, List<? super T>)`.

In (b), the arguments in the call are `(List<Number>, List<Integer>)`. The actual type parameter is `Number`. The arguments do not satisfy the formal parameters `(List<? extends Number>, List<? super Number>)`. `List<Number>` is a subtype of `List<? extends Number>`, but `List<Integer>` is not a subtype of `List<? super Number>`.

In (c), the arguments in the call are `(List<Number>, List<Integer>)`. The actual type parameter is `Integer`. The arguments do not satisfy the formal parameters `(List<? extends Integer>, List<? super Integer>)`. `List<Number>` is not a subtype of `List<? extends Integer>`, although `List<Integer>` is a subtype of `List<? super Integer>`.

In (d), the arguments in the call are `(List<Integer>, List<Number>)`. The inferred type is `Integer`. The arguments satisfy the formal parameters `(List<? extends Integer>,` `List<? super Integer>)`.

In (e), the arguments in the call are `(List<Integer>, List<Number>)`. The actual type parameter is `Number`. The arguments satisfy the formal parameters `(List<? extends Number>, List<? super Number>)`.

In (f), the arguments in the call are `(List<Integer>, List<Number>)`. The actual type parameter is `Integer`. The arguments satisfy the formal parameters `(List<? extends Integer>, List<? super Integer>)`.

11.10 *(f)*

(a) invokes the zero-argument constructor at (1). (b) invokes the constructor at (2) with `T` as `String` and `V` as `String`.

(c) invokes the constructor at (2) with `T` as `String` and `V` as `Integer`.

(d) invokes the constructor at (3) with `T` as `Integer` and `V` as `String`.

(e) invokes the constructor at (3) with `T` as `String` and `V` as `Integer`.

(f) cannot infer type arguments for `Box<>`. From the constructor call signature `(String, Integer)` one would assume that `T` was `String` and `V` was `Integer`. The parameterized type `Box<Integer>` of the reference on the left-hand side implies `T` is `Integer`, which contradicts that `T` is `String` on the right-hand side.

11.11 *(b)*

It is the fully qualified name of the class after erasure that is printed at runtime. Note that it is the type of the object, not the reference, that is printed. The erasure of all the lists in the program is `ArrayList`.

11.12 *(e)*

(a) contains incompatible types for assignment in the `main()` method. The method will return a `Collection` whose element type is some unknown subtype of `CharSequence` (`Collection<? extends CharSequence>`). As it is not known which subtype, assignment to `Collection<String>` cannot be allowed.

(b) contains an incompatible return value in the `delete4LetterWords()` method. The declared return type is `List<E>` but the `return` statement returns a `Collection<E>`. It cannot convert from `Collection<E>` to `List<E>`.

In (c), the reference `words` denotes a `Collection` whose

element type is some unknown subtype of `CharSequence` (`Collection<? extends CharSequence>`). In the `for(:)` loop, the loop variable `word` is of type `E`. The unknown element type of `words` cannot be converted to `E`.
(d) contains an incompatible return value in the `delete4LetterWords()` method: It cannot convert from `Collection<E>` to `List<E>`, as explained in (b). In the `for(:)` loop, the unknown element type of `words` cannot be converted to an element of type `E`, as explained in (c). (e) is OK.
In (f), the keyword `super` cannot be used in a constraint. It can only be used with a wildcard (`?`).

11.13 (b) and (f)

After erasure, the method at (1) has the signature `overloadMe(List, List)`. Since all methods are declared `void`, they must differ in their parameter list after erasure in order to be overloaded with the method at (1). All methods have different parameter lists from that of the method at (1), except for the declarations (b) and (f). In other words, all methods have signatures that are not override equivalent to the signature of the method at (1), except for (b) and (f).

11.14 (b)

Passing or assigning a raw list to either a list of `Integer`s or to a list of type parameter `T` is not type-safe. Passing or assigning a raw `List` to a `List<?>` is always permissible.

11.15 (c), (f), (i), and (k)

The type parameter `N` in `SubC1` does *not* parameterize the supertype `SupC`. The erasure of the signature at (3) is the same as the erasure of the signature at (1) (i.e., it is a name clash). Therefore, of the three alternatives (a), (b), and (c), only (c) is correct. The type parameter `N` in `SubC1` cannot be guaranteed to be a subtype of the type parameter `T` in `SupC`—that is, incompatible return types for the `get()` methods at (4) and (2), which are not overridden. Also, methods cannot be overloaded if only return types are different. Therefore, of the three alternatives (d), (e), and (f), only (f) is correct.
The type parameter `N` in `SubC2` is a subtype of the type parameter `M`, which parameterizes the supertype `SupC`. The erasure of the signature at (5) is still the same as the erasure of the signature at (1) (i.e., it is a name clash). Therefore, of the three alternatives (g), (h), and (i), only (i) is correct.
The type parameter `N` in `SubC1` is a subtype of the type

parameter `T` (through `M`) in `SupC`—that is, covariant return types for the `get()` methods at (6) and (2), which are overridden. Therefore, of the three alternatives (j), (k), and (l), only (k) is correct.

11.16 *(a), (c), and (e)*

In (a), because of the way an enum type `E` is implemented as a subtype of the `java.lang.Enum<E>` class in Java, we cannot define a generic enum type.

In (c), generic exceptions or error types are not allowed because the exception handling mechanism is a runtime mechanism and the JVM is oblivious to generics.

In (e), anonymous classes do not have a name, but a class name is needed for declaring a generic class and specifying its formal type parameters. A *parameterized* anonymous class can always to declared.

11.17 *(d)*

Casts are permitted, as at (2) through (6), but can result in an unchecked warning. The *assignment* at (5) is from a raw type (`List`) to a parameterized type (`List<Integer>`), resulting in an unchecked assignment conversion warning. Note that at (5) the cast does not pose any problem. It is the assignment from generic code to legacy code that can be a potential problem, and flagged as an unchecked warning.

At (6), the cast is against the erasure of `List<Integer>`—that is, `List`. The compiler cannot guarantee that `obj` is a `List<Integer>` at runtime, and therefore flags the cast with an unchecked warning.

Only reifiable types in casts do not result in an unchecked cast warning.

11.18 *(e)*

Instance tests in the `scuddle()` method use the reified type `List<?>`. All assignments in the `main()` method are type-safe.

11.19 *(c)*

The erasure of `E[]` in the method `copy()` is `Object[]`. The array type `Object[]` is actually cast to `Object[]` at runtime—that is, an identity cast. The method `copy()` returns an array of `Object`. In the `main()` method, the assignment of this array to an array of `String`s results in a `ClassCastException`.

11.20 *(e)*

The method header at (1) is valid. The type of the variable arity parameter can be generic. The type of the formal parameter `aols` is an array of `List`s of `T`. However, the compiler issues a potential heap pollution warning because of variable arity parameter `aols`.

The `main()` method at (2) can be declared as `String...`, as it is equivalent to `String[]`, but no potential heap pollution warning is issued, as it is a reifiable type. The statement at (3) creates an array of `List`s of `String`s. However, the compiler issues an unchecked conversion warning, since a raw type (`List[]`) is being assigned to a parameterized type (`List<String>[]`).

The formal type parameter `T` is inferred to be `String` in the method call at (4). The method `doIt()` prints each list in its variable arity parameter `aols`.

12 Collections, Part I: ArrayList<E>

12.1 *(e)*

The `for(;;)` loop correctly increments the loop variable so that all the elements in the list are traversed. Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

12.2 *(b) and (c)*

In the method `doIt1()`, one of the common elements (`"Ada"`) between the two lists is reversed. The value `null` is added to one of the lists but not the other.

In the method `doIt2()`, the two lists have common elements. Swapping the elements in one does not change their position in the other.

12.3 *(c)*

The element at index 2 has the value `null`. Calling the `equals()` method on this element throws a `NullPointerException`.

12.4 *(f)*

Deleting elements when iterating over a list requires care, as the size changes and any elements to the right of the deleted element are shifted left. Incrementing the loop variable after deleting an element will miss the next element (i.e., the last occurrence of `"Bob"`). Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

12.5 *(f)*

The `while` loop will execute as long as the `remove()` method returns `true`—that is, as long as there is an element with the value `"Bob"` in the list. The `while` loop body is the empty statement. The `remove()` method does not throw an exception if an element value is `null`, or if it is passed a `null` value.

12.6 *(b)*

An `ArrayList` object is populated with the content from the `String` array. Just like with an array, an array list has a 0-based index. The item at index 1 in this array list is replaced with the string `"X"`, making this array list content `[A,X,B,A]`. Then a new item is added at the same index position, causing all other items in the list to be shifted by one position, making this array list content `[A,X,X,B,A]`. Lastly, an item at index 2 is removed, giving the result `[A,X,B,A]`.

12.7 *(a)*

The method `Arrays.asList()` creates a fixed-size list, which does not allow items to be added or removed, but its content can be changed, which is what the `set()` operations do, replacing items at index 1 and 2 with `"X"`.

12.8 *(c)*

The two arrays and the list in the `main()` method contain references to the same `Song` objects. These are not independent copies, so modifications on a shared `Song` object will be visible no matter how this object is accessed.

12.9 *(b)*

A list that is created using the `List.of()` method shares the elements with the original array. However, changes applied to the original array are not reflected in the list.

12.10 *(a)*

The method `toArray()` returns an array with all the elements in the list. The type of the array is given by the array passed as a parameter. If the length of the argument array is equal to the size of the list, the argument array is used. The argument array is also used if its length is greater than the size of the list, but after copying the elements to the array, the remaining elements in the array are filled with `null` values. Otherwise, a new array of appropriate size is created. In the sample code, the length of the array is equal to the size of the list. Therefore, the argument array is used. Afterwards, the lowercase version of the element at index 0 in the original list is assigned to the element at index 1 in the array.

12.11 *(b)*

An empty `ArrayList` object is created to store `Character` objects, using a constructor with a capacity of 3. Five `char` values from `'a'` to `'e'` are boxed as `Character` objects and added to this list. Remember that a list auto-expands its capacity as required.

13 Functional-Style Programming

13.1 *(e)*

A functional interface can be implemented by lambda expressions and classes. A functional interface declaration can only have one abstract method declaration. In the body of a lambda expression, all members in the enclosing class can be accessed. In the body of a lambda expression, only final or effectively final local variables in the enclosing scope can be accessed.

13.2 *(e), (f), (g), and (i)*

The assignments at (5), (6), (7), and (9) will not compile. We must check whether the function type of the target type and the type of the lambda expression are compatible. The function type of the target type `p1` in the assignment statements from (1) to (5) is `String -> void` (i.e., a `void` return). The function type of the target type `p2` in the assignment statements from (6) to (10) is `String -> String` (i.e., a non-`void` return). Below, the functional type of the target type is shown in a comment with the prefix `LHS` (left-hand side), and the type of the lambda expression for each assignment from (1) to (10) is shown in a comment with the prefix `RHS` (right-hand side).

Click here to view code image

```
    Funky1 p1;                                     //      LHS: String -> voi
    p1 = s -> System.out.println(s);               // (1) RHS: String -> voi
    p1 = s -> s.length();                          // (2) RHS: String -> int
    p1 = s -> s.toUpperCase();                     // (3) RHS: String -> Stri
    p1 = s -> { s.toUpperCase(); };                // (4) RHS: String -> voi
//  p1 = s -> { return s.toUpperCase(); };         // (5) RHS: String -> Stri
```

```
      Funky2 p2;                                //     LHS: String -> Stri
   // p2 = s -> System.out.println(s);          // (6) RHS: String -> voi
   // p2 = s -> s.length();                     // (7) RHS: String -> int
      p2 = s -> s.toUpperCase();                // (8) RHS: String -> Stri
   // p2 = s -> { s.toUpperCase(); };           // (9) RHS: String -> voi
      p2 = s -> { return s.toUpperCase(); };    // (10)RHS: String -> Stri
```

Remember that the non-`void` return of a lambda expression with an *expression statement* as the body can be interpreted as a `void` return, if the function type of the target type returns `void`. This is the case at (2) and (3). The return value is ignored. The type `String -> String` of the lambda expression at (5) is not compatible with the function type `String -> void` of the target type `p1`.
The type of the lambda expression at (6), (7), and (9) is not compatible with the function type `String -> String` of the target type `p2`.

13.3 *(d)*

The three interfaces are functional interfaces. `AgreementB` explicitly provides an abstract method declaration of the `public` method `equals()` from the `Object` class, but such declarations are excluded from the definition of a functional interface. Thus `AgreementB` effectively has only one abstract method. A functional interface can be implemented by a concrete class, such as `Beta`. The function type of the target type in the assignments (1) to (3) is `void -> void`. The type of the lambda expression at (1) to (3) is also `void -> void`. The assignments (1) to (3) are legal.
The assignment at (4) is legal. Subtype references are assigned to supertype references. References `o`, `a`, and `c` refer to the lambda expression at (3).
The assignment at (5) is legal. The reference `b` has the type `AgreementB` and class `Beta` implements this interface.
The code at (6), (7), and (8) invokes the method `doIt()`. The code at (6) evaluates the lambda expression at (3), printing `Jingle|`. The code at (7) invokes the `doIt()` method on an object of class `Beta`, printing `Jazz|`. The code at (8) also evaluates the lambda expression at (3), printing `Jingle|`.
At (9), the reference `o` is cast down to `AgreementA`. The reference `o` actually refers to the lambda expression at (3), which has target type `AgreementC`. This interface is a subtype of `AgreementA`. A subtype is cast to a supertype, which is allowed, so no `ClassCastException` is thrown at runtime. Invoking the `doIt()` method again results in evaluation of the

lambda expression at (3), printing `Jingle|`.

Apart from the declarations of the lambda expressions, the rest of the code is plain-vanilla Java. Note also that the following assignment that defines a lambda expression would not be valid, since the `Object` class is not a functional interface and therefore cannot provide a target type for the lambda expression:

Click here to view code image

```
Object obj = () -> System.out.println("Jingle"); // Compile-time error!
```

13.4 *(c)*

The method `removeIf()` accepts as an argument a lambda expression that implements a `Predicate<E>` interface. This method removes all strings of length 3 from the list. The `for (:)` loop calculates the sum of the lengths of the remaining strings in the list, producing a result of 9.

13.5 *(c)*

The method `removeIf()` accepts a lambda expression that first converts a string to lowercase and then tests whether the resulting string starts with the character `'a'`. Note that the predicate only performs the test, and it does not actually modify the strings in the list. Only the strings `"ANNA"` and `"ALICE"` pass the test and are removed.

13.6 *(i)*

The lambda expression uses identifier `s` as a parameter name, which is illegal because a variable called `s` is already defined in the enclosing context of the lambda expression.

13.7 *(c)*

There are two predicates defined in this code. The first predicate determines whether a string contains the letter `O`, and the second one determines whether a string ends with the letter `P`. The composed predicate `filter1.and(filter2).negate()` determines whether a string does *not* contain an `O` *or* it does *not* end with a `P`. Only the strings `"PLOT"` and `"LEAP"` pass this test and are removed from the list by the `removeIf()` method, leaving only the strings `"FLOP"` and `"LOOP"` in the list.

13.8 *(d)*

The `compose()` method is inherited by the `UnaryOperator<T>` from its superinterface `Function<T, T>`. This method returns a

`Function<T, T>`. As an instance of a super-type (`Function<T, T>`) cannot be assigned to a subtype (`UnaryOperator<T>`), the assignment to `f3` results in a compile-time error.

13.9 (b)

All `String` values in the list are replaced with their lowercase equivalents using the `replaceAll()` method which accepts a lambda expression that implements a `UnaryOperator<String>`. The two consumers are applied to the values in this `List`.
Consumer `c1` is changing the first letter of every string in the list to uppercase, but it does not replace actual `String` objects stored within this list. Next, consumer `c2` prints the content of this list, which has been produced by the `replaceAll()` method.

13.10 (b)

Regarding method references, the method `isEven()` is static and therefore should be referred to using the class name `Test`, while the method `printValue()` is an instance method, and therefore should be referred to using a reference of the class `Test`.

13.11 (a)

The target reference for the bounded instance method reference is set explicitly. The unbounded instance method reference interprets the first argument as the target reference.

13.12 (d)

Notice that the `BiFunction` in this example is using raw type. Therefore, the `x` and `y` parameters are of the `Object` type. This means that a division operator cannot be applied in this case.

13.13 (a)

Functions `f1` and `f2` are combined to concatenate the prefix and the postfix around the value supplied to the `apply()` method argument. Notice that conversion to `String` works for any object in Java. Therefore, `Function` objects can use raw types.

14 Object Comparison

14.1 (b) and (d)

It is recommended that (a) is fulfilled, but it is not a requirement. (c) is also not required, but such objects will lead to collisions in the hash table, as they will map to the same bucket.

14.2 *(a), (b), (d), and (h)* (c) is eliminated, since the `hashCode()` method cannot claim inequality if the `equals()` method claims equality. (e) and (f) are eliminated, since the `equals()` method must be reflexive, and (g) is eliminated, since the `hashCode()` method must consistently return the same hash code during execution.

14.3 *(b), (d), and (e)* (a) and (c) fail to satisfy the properties of an equivalence relation. (a) is not transitive, and (c) is not symmetric.

14.4 *(a) and (e)* (b) is not correct, since it will throw an `ArithmeticException` when called on a newly created `Measurement` object. (c) and (d) are not correct, since they may return unequal hash codes for two objects that are equal according to the `equals()` method.

14.5 *(c)*

The generic static method `cmp()` returns a comparator (implemented as a lambda expression) that reverses the natural ordering of a `Comparable` type. The natural ordering of the class `Person` is ordering by `name` first and then by `age`, using the reverse comparators `strCmp` and `intCmp`. `p1` is *less* than `p2` because of `name`, and `p1` is *greater* than `p3`, because of `age`, as their names are equal.

14.6 *(d)*

All methods implement reverse natural ordering, except the method at (4). The method reference `Comparable::compareTo` is equivalent to the lambda expression `(e1, e2) -> e1.compareTo(e2)`—that is, natural ordering.

14.7 *(b)*

A lambda expression that implements the `Comparator<String, String>` is used to sort the array in ascending order based on the text representation of `Integer` objects. Basically, each array element is converted to a `String` before it is compared. The ordering is that of `String` objects, where `"-23"` is less than `"-41"` lexicographically.

14.8 *(a)*

The lambda expression that implements the `Comparator<Album>` interface defines a total ordering of `Albums` based on the difference between the lengths of the album titles, resulting in the list being sorted in ascending order by `title` length. The resulting list is then printed using the lambda expression that implements the `Consumer<Album>` interface.

14.9 (b)

The `equals()` method of the `Album1` class checks whether the object is not `null` and of the same type as the current object before comparing album titles. This is a strict check that verifies whether the object with which the current object is compared is of exactly the same type, using the following condition: `(getClass() != obj.getClass())`. Alternatively, a less strict check that allows type substitution is also possible: `(obj instanceof Album1)`. The difference between these two approaches is that the `instanceof` operator can return `true` when comparing this object to another object that is an instance of the subtype. Of course, this would not be the case if specific class types are compared.

Note that the logic in the `main()` method compares an `Album1` to an `LP`, which is actually a subclass of `Album1`. This means that even though both of these objects have the same title, they would not be considered equal because the logic of the `equals()` method implements a strict type comparison.

14.10 (b) and (d)

The `Comparator<A>` interface defines the `compare()` method that is designed to compare two argument objects of class `A` to establish their ordering. Each `Comparator<A>` implementation can define a different total ordering for the objects.

15 Collections: Part II

15.1 (a)

The expression in the `for(:)` loop header (in this case the call to the `makeCollection()` method) is only evaluated once.

15.2 (c) and (d)

The `for(:)` loop does not allow the list to be modified structurally. In (a) and (b), the code will throw a `java.util.ConcurrentModificationException`. Note that the iterator in (d) is less restrictive than the `for(:)` loop, allowing elements to be removed in a controlled way.

15.3 (d)

The iterator implemented will iterate over the elements of the list in the reverse order, and so will the `for(:)` loop. The `Iterable<E>` and the `Iterator<E>` interfaces are implemented correctly. Note that the anonymous class that implements the iterator is parameterized by the formal type parameter `T` of the generic class `AnotherListIterator<T>`.

15.4 (b) and (d)

Some operations on a collection may throw an `UnsupportedOperationException`. This exception type is unchecked, and the user code is not required to explicitly handle unchecked exceptions. A `List<E>` allows duplicate elements. An `Array-List<E>` is implemented using a resizable array. The capacity of the array will be expanded automatically when needed. The `Set<E>` allows at most one `null` element.

15.5 (d)

The program will compile without error, and will print all primes below 25 at runtime. All collection implementations used in the program implement the `Collection<E>` interface. The implementation instances are interchangeable when denoted by `Collection` references. None of the operations performed on the implementations will throw an `UnsupportedOperationException`. The program finds the primes below 25 by removing all values divisible by 2, 3, and 5 from the set of values from 2 through 25.

15.6 (b)

The `remove()` method removes the last element returned by either the `next()` or `previous()` method. The four `next()` calls return `A`, `B`, `C`, and `D`. `D` is subsequently removed. The two `previous()` calls return `C` and `B`. `B` is subsequently removed.

15.7 (c), (d), (e), and (f)

Sets cannot have duplicates. `HashSet<E>` does not guarantee the order of the elements in (a) and (b), so there is no guarantee that the program will print `[1, 9]`. Because `LinkedHashSet<E>` maintains elements in insertion order in (c) and (d), the program is guaranteed to print `[1, 9]`. Because `TreeSet<E>` maintains elements sorted according to the natural ordering in (e) and (f), the program is guaranteed to print `[1, 9]`.

15.8 (c) and (d)

The output from each statement is shown below. (a) `[sea, shell, soap]`

(b) `[sea, shell]`

(c) `[soap, swan]`

(d) `[swan]`

(e) `[shell, soap]`

(f) `[sea, shell]`

15.9 (b) and (d)

Although all *keys* in a map must be unique, duplicate *values* can occur. Since values are not unique, the `values()` method returns a `Collection<V>` and not a `Set<V>`. The collections returned by the `keySet()`, `entrySet()`, and `values()` methods are backed by the underlying map. This means that changes made in one are reflected in the other. Although implementations of the `SortedMap<K, V>` interface maintain the entries sorted according to key-sort order, this is not a requirement for classes that implement the `Map<K, V>` interface. For instance, the entries in a `HashMap<K, V>` are not sorted.

15.10 (a), (c), and (d)

The key of a `Map.Entry<K, V>` cannot be changed, since the key is used for locating the entry within the map. There is no `set()` method. The `setValue()` method is optional.

15.11 (b)

A set is a collection of unique elements, so an attempt to insert the same element twice is ignored, with no exception raised. The ordering of elements in the set is determined by the `Comparator<E>` passed to the `TreeSet` constructor. The comparator passed compares the element strings in the *reverse natural ordering*.

15.12 (b)

The `set1` object sorts elements according to the *reverse natural ordering*. The `set2` object retains that ordering. In the statement

Click here to view code image

```
NavigableSet<String> set2 = new TreeSet<>(set1);
```

the signature of the constructor called is the following:

Click here to view code image

```
TreeSet<String>(SortedSet<String> set)
```

resulting in the same ordering for the elements in `set2` as in `set1` (i.e., reverse natural ordering). Note that class `NavigableSet<E>` is a subclass of class `SortedSet<E>` class.

15.13 (a)

The `set1` object sorts the elements according to reverse natural ordering. In the statement

Click here to view code image

```
NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
```

the signature of the constructor called is

Click here to view code image

```
TreeSet<String>(Collection<? extends String> collection)
```

resulting in the elements in `set2` being sorted according to *natural ordering*, and not according to the reverse natural ordering of `set1`.

15.14 (a)

The `set1` object sorts its elements in reverse natural ordering. It is polled from the tail, so its elements are fetched according to natural ordering. On the other hand, the elements in `set2` are sorted according to natural ordering. `set2` is polled from the head, so its elements are fetched according to natural ordering.

15.15 (b)

A map view method creates half-open intervals (i.e., the upper bound is not included), unless the inclusion of the bounds is explicitly specified. Clearing a map view clears the affected entries from the underlying map. The argument to the `sumValues()` method can be any subtype of `Map<K, V>`, where the type of the value is `Integer`.

15.16 (b), (e), and (f) (a) throws a `ConcurrentModificationException`. We cannot remove an entry in a `for(:)` loop. (c) throws a `ConcurrentModificationException` as well, even though we use an iterator. The `remove()` method is called on the map, not on the iterator. The argument to the `remove()` method of the map must implement `Comparable`. `Map.Entry<K, V>` does not, resulting in a `ClassCastException` in (d).

We can remove an entry from the underlying map when iterating over the key set using an iterator, as in (b). (e) creates a map view of one entry and clears it, thereby clearing it also from the underlying map. (f) removes the entry for `"Shampoo"` from the map, since the lambda expression returns the value `null`.

15.17 (e)

The variable `sumVal` is not effectively `final` when referenced in the lambda expression body, as it is incremented for each entry in the map.

15.18 (c)

The `computeIfAbsent()` method returns an empty `TreeSet` if the key is not found in the map. If the key is found, it returns the associated `TreeSet`. The `add()` method is invoked on the `TreeSet` that is returned by the `computeIfAbsent()` method. The `add()` method adds its argument to this `TreeSet`. The resulting map is a *multimap*— that is, a key can be associated with a collection of values.

15.19 (b)

The `BiFunction<Integer, String, String>` implemented by the lambda expression computes a new value for the key of an entry in the map. The `switch` statement determines the new value based on the key. The lambda expression returns the value `"FIRST"` for key 1, the value `"SECOND"` for key 2, and so on. The `replaceAll()` method replaces the value of each entry in the map with the new value computed for the key by the `BiFunction<T, U, R>`.

15.20 (b)

The class `StringBuilder` implements the `Comparable<E>` interface. The `sort()` method sorts the elements in reverse natural ordering: `[C, B, A]`. The method `sublist()` returns the elements in the open interval `[1, 2)`—that is, the element at index `1`, which is `"B"`.

15.21 (b), (c), (f), and (g)

The `Collections.addAll()` method adds the elements to an existing list when it is called. All three elements are in `list1` when (1) is executed. The `Arrays.asList()` method returns a new list every time it is called. Only the string `"Howdy"` is in `list2` when (2) is executed. The `Collection.addAll()` method adds the elements of its argument collection to the collection on which it is called. In this case, `list3` has the same elements as `list1`. Calling the constructor with a collection as an argument initializes the new list with the elements of the specified collection. In this case, `list4` has the same elements as `list2`.

Creating a new list by calling the constructor with a collection as an argument returns an `ArrayList` initialized with the elements of the collection.

15.22 (a) and (f)

The largest value a match can return is the largest index—that is, *array.length – 1* (`==3`). The key must be equal to the largest element in the array. If no match is found, a negative value is returned, which is computed as follows: *– (insertion point + 1)*. The smallest value is returned for a key that is greater than the largest element in the array. This key must obviously be placed at the index *array.length (= =4)*, after the largest element—that is, the insertion point is 4. The value of the expression *– (insertion point + 1)* is `-5`, which is the smallest value printed by the method.

15.23 (c)

The operation `pollFirst()` does not throw an exception, but rather returns `null` when the `Deque` object is empty. The operations `peekFirst()` and `peekLast()` return the first and last elements from the `Deque` object, respectively, but do not remove elements from the `Deque`. The operations `pollFirst()` and `pollLast()` return the first and last elements from the `Deque` object, respectively, and remove them from the `Deque`. The operation `offerFirst()` inserts elements at the head of the `Deque`. The operation `offerLast()` inserts elements at the tail of the `Deque`.

15.24 (b)

A set cannot have duplicates. This means that object `x` was only added once to the set. The `add()` method does not throw an exception, but rather returns `false` when an element cannot be added to the set.

15.25 (d)

The first two add operations result in the list `[1, 2]`. Next, a `null` value is inserted at index 2, and the value `3` is inserted at index 3, which results in the list `[1, 2, null, 3]`. Next, the value `4` is inserted at index 2, shifting elements towards the end of the list, resulting in the list `[1, 2, 4, null, 3]`. The element at index 2 is replaced with the value `3`, giving the list `[1, 2, 3, null, 3]`, and then the element at index 2 is removed, giving the list `[1, 2, null, 3]`. Finally, the value `2` is inserted at index 2, which results in the list `[1, 2, 2, null, 3]`.

16 Streams

16.1 *(b)*

The `mapToInt()` operation converts a `Stream<String>` to an `IntStream` whose elements are the length of the strings in its input stream. The `int` stream will contain the values `1`, `3`, `2`, and `4`, corresponding to the length of the strings. The `filter()` operation discards strings of length `4`. Its output stream will only contain the values `1`, `3`, and `2`. The `reduce()` method performs a functional reduction, starting with the initial value of `1`. Its accumulator multiplies the cumulative result with the current value in the `int` stream, with the computation proceeding as follows:

```
(x, y) -> x * y
(1, 1) -> 1 * 1 => 1
(1, 3) -> 1 * 3 => 3
(3, 2) -> 3 * 2 => 6
```

16.2 *(d)*

The `filter()` intermediate operation is designed to return a stream whose elements match the given `Predicate`. The `findFirst()` terminal operation does not necessarily return the first element from the stream when this stream is processed in parallel mode. The `reduce()` terminal operation performs a functional reduction on the elements of the stream, and it uses an accumulator and not a `Predicate`. The `sorted()` intermediate operation sorts the elements according to their natural order, or according to the total order specified by a `Comparator`.

16.3 *(d) and (e)*

(a) performs a functional reduction starting with the initial value `0` and adding all values in the stream to compute the sum of the values.
(b) performs the same functional reduction as in (a) but in parallel mode.
(c) performs the same functional reduction as in (a), but does

not use the initial value of `0`. It uses the value of the first element in the stream, if there is one. Since the stream can be empty, it returns an `OptionalInt` object. The `orElse()` operation on this `OptionalInt` object retrieves an `int` value if it has one; otherwise, the operation returns the value `0`.
(d) uses `0` as the initial value, which means that this value will be returned if the steam is empty. Therefore, the operation is guaranteed to return an `int` value, and not an `OptionalInt`. The `orElse()` operation cannot be invoked on an `int` value, so this code will not compile.
(e) refers to the variable `sum` within the lambda expression. As it has not been initialized, the code will fail to compile. Note that only final or effectively final variables can be referenced within a lambda expression.
(f) computes the sum of all values in the stream.

16.4 *(b) and (d)*

The stream will contain the following values: `0`, `1`, `2`, `3`, and `4`. Note that `x` designates the cumulative value and `y` designates the current element. (a) performs functional reduction using the identity value `0` as the initial value and the accumulator adds 1 to the cumulative result for each element. The reduction proceeds as follows:

```
(x, y) -> x + 1
(0, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(b) performs a similar functional reduction as in (a), but uses the value of the first element (`0`) as the initial value. So it would result in one addition less than (a). The reduction proceeds as follows:

```
(x, y) -> x + 1
(0, 1) -> 0 + 1 => 1
(1, 2) -> 1 + 1 => 2
(2, 3) -> 2 + 1 => 3
(3, 4) -> 3 + 1 => 4
```

(c) performs a functional reduction similar to (a), but now the accumulator increases the value of the stream element `y` by 1. The reduction proceeds as follows:

```
(x, y) -> y + 1
(0, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
```

```
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(d) performs a functional reduction using the initial value `0` and where the accumulator returns the value of the stream element `y`. The reduction proceeds as follows:

```
(x, y) -> y
(0, 0) -> 0 => 0
(0, 1) -> 1 => 1
(1, 2) -> 2 => 2
(2, 3) -> 3 => 3
(3, 4) -> 4 => 4
```

(e) performs a function reduction which is similar to (c), except that it uses the identity value of `1` as the initial value. The reduction proceeds as follows:

```
(x, y) -> y + 1
(1, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(f) performs a functional reduction of the stream elements using the `count()` operation, which in this case results in the value `5`.

16.5 (d)

(a) produces three groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream. The `Predicate` expression discards any value containing the string `"C"`.

```
1 []
2 [AA, DD]
3 [BBB, EEE]
```

(b) produces three groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream. The `filter()` operation discards all values except those that contain the string `"C"`.

```
1 [C]
2 []
3 []
```

(c) produces two groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream.

However, the `filter()` operation discards any values containing the string `"C"`, before the groups are created.

```
2 [AA, DD]
3 [BBB, EEE]
```

(d) results in a single group based on the `Integer` values corresponding to the length of `String` objects in the stream. The `filter()` operation discards any values except those that contain the string `"C"`, before any groups are created.

```
1 [C]
```

16.6 *(d)*

An infinite stream of string `"A"` is generated. The first `peek()` operation prints the string `"B"`. The `Predicate` of the `takeWhile()` operation returns `false` immediately on encountering the first element in the stream which is `"A"`. The `takeWhile()` operation only takes an element if it is not `"A"`. It short-circuits the stream processing, resulting in an output stream that is empty. The `Consumer` of the second `peek()` operation does not execute, as the stream is empty. The `anyMatch()` terminal operation returns `false` on encountering an empty stream.

16.7 *(a)*

A stream of `int` values that correspond to character codes for letters `'a'`, `'b'`, `'c'`, and `'d'` is generated. These values are mapped to single-letter strings that are converted to uppercase. The `filter()` operation discards a letter if it does not match a vowel, which results in an output stream with only the element `"A"`, which is printed.

16.8 *(b) and (c)*

A stream of `int` values 0, 1, 2, 3, and 4 is generated. The `filter()` operation discards all even numbers from this stream, retaining only the odd numbers 1 and 3, which are then printed.

(a) generates a stream of `int` values 0, 1, 2, 3, 4, and 5 which is one value more than in the program. Even numbers are discarded, retaining only the odd numbers 1, 3, and 5 which are then printed.

(b) generates a stream of `int` values between 0 and 10. The `takeWhile()` operation only takes values less than 5. It truncates the stream when the element is greater than or equal to 5. The `filter()` operation discards all even numbers from the truncated stream, retaining only the odd numbers 1 and 3, which are then printed.

(c) generates a stream of `int` values between 0 and 10, which is then truncated to the first five values. The `filter()` operation discards all even numbers from this truncated stream, retaining only the odd numbers 1 and 3, which are then printed.

(d) generates an infinite stream of 0s. The expression `x++` will always evaluate to 0, when `x` is initialized to 0. The `takeWhile()` operation will continue to take elements from the stream, as its `Predicate` will always return `true`. The `filter()` operation will continue to discard each element, as its value will always be `0`. The terminal operation will never get to process an element. This state of affairs will continue indefinitely, with nothing being printed.

(e) does not compile because the variable `x` referred to in the lambda expression is not final. The expression `x++` will change the value in `x`, which is not permitted.

16.9 *(d)*

Two streams of `String` objects containing the values `"A"`, `"B"`, `"C"` and the values `"X"`, `"Y"`, `"Z"` are concatenated into a single stream. The resulting stream has the values `"X"`, `"Y"`, `"Z"`, `"A"`, `"B"`, `"C"`.

The functional reduction concatenates the elements from this new stream into a single string. This operation returns an `Optional<String>`, as the reduction uses the first element as the initial value. The result string in the `Optional<String>` is returned by the `get()` method of the `Optional` class.

Note that `a` denotes the cumulative result and `b` denotes the current element in the stream. The reduction operation is performed as follows:

Click here to view code image

```
(a,          b)   ->  b  +  a
("X",       "Y") -> "Y" + "X" => "YX"
("YX",      "Z") -> "Z" + "YX" => "ZYX"
("ZYX",     "A") -> "A" + "ZYX" => "AZYX"
("AZYX",    "B") -> "B" + "AZYX" => "BAZYX"
("BAZYX",   "C") -> "C" + "BAZYX" => "CBAZYX"
```

16.10 *(a)*

All process a stream of `String` objects that are one-letter strings from `"A"` to `"E"`. Grouping is done based on a classifier which is a `Function`, whereas partitioning is done based on a `Predicate`. Identical lambda expressions implement the classifier and the predicate in all options. The lambda expression returns `true` if the single-letter string is a vowel. The map created by both operations will have the type `Map<Boolean, List<String>>`, where the keys are `Boolean` and the value associated with a key is a `List<String>`. The list is created implicitly, as in (a) and (b), or explicitly in a downstream collector, as in (c) and (d).

The filtering is done by the same predicate in all options, discarding any one-letter string that is greater than the string `"A"`. Effectively, the only element processed by the stream is the string `"A"`.

The `partitioningBy()` operation always creates entries for the `Boolean.TRUE` and `Boolean.FALSE` keys in the result map, even if no values can be computed for these keys from the stream elements. On the other hand, the `groupingBy()` operation creates entries for keys computed by its classifier—that is, an entry is created for the key `Boolean.TRUE` or `Boolean.FALSE` depending on the elements in the stream. However, when the `filtering()` operation is used as a downstream collector in the grouping operation, entries for both the `Boolean.TRUE` and `Boolean.FALSE` keys are created, regardless of whether any value is associated with these keys.

In (a), grouping creates only one entry for the `Boolean.TRUE` key in the result map based on its `Predicate` being `true`, since the only element `"A"` in the stream is a vowel. The output is the following:

```
true [A]
```

In (b), partitioning creates two entries in the result map: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants). The only element `"A"` is associated with the `true` key as it a vowel. The output is the following:

```
false [] true [A]
```

In (c), grouping creates two entries: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants), as its downstream collector is a `filtering()` operation. Since the string `"A"` is a vowel, it is accumulated in the list associated with the `Boolean.TRUE` key. The list associated with the `Boolean.FALSE` key remains empty. The output is the following:

```
false []
true [A]
```

In (d), partitioning creates two entries: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants), regardless of its downstream collector. Since the string `"A"` is a vowel, it is accumulated in the list associated with the `Boolean.TRUE` key. The list associated with the `Boolean.FALSE` key remains empty. The output is the following:

```
false []
true [A]
```

This means that (a) resulted in only one entry in the map, while the other resulted in two identical entries.

16.11 (d)

It is important to note that the stream of strings is not processed separately from the stream of chars, but rather they are fused into a single stream pipeline. This is because only one terminal operation exists in the stream pipeline. This means that the parallel processing applies to the entire pipeline. The `sort()` operation sorts the characters in the flattened stream, but the `forEach()` operation cannot be relied upon to respect the order, especially in a parallel stream. The `forEachOrdered()` operation will give a deterministic result regardless of the execution mode of a stream. The result from the program is therefore unpredictable.

16.12 (c) and (d)

The `filter()` method accepts a `Predicate`. The methods `peek()` and `forEach()` accept a `Consumer`. `map()` accepts a `Function`, `max()` accepts a `Comparator`, and `findAny()` does not accept any parameters.

16.13 (b) and (f)

The methods `peek()`, `map()`, `filter()`, and `sorted()` are all intermediate operations. The methods `forEach()` and `min()` are terminal operations.

16.14 (b) and (d)

Short-circuit methods may produce finite results for potentially an infinite stream. For example, the operations `limit()` and `anyMatch()` are short-circuit operations. A short-circuit operation terminates the stream pipeline, whether or not all elements in the stream have been processed.

16.15 *(f)*

These statements all perform an equivalent functional reduction of counting the number of elements in the stream. Empty string or `null` elements are still counted as elements. Thus all of these operations return the value `6`. Counting the number of elements in the stream can be achieved using the `count()` method of the `Stream` interface, or the `counting()` method provided by the `Collectors` class. Another solution is to map all stream elements to the value `1`, and then summing up the `1`s will give the same result.

16.16 *(b) and (d)*

In (a), a set of `String` objects is constructed that contains the values `"XX"`, `"XXXX"`, `""`, and `"X"` because the `filter()` method removes all `null` elements from the stream. Notice the absence of the duplicate values due to the fact that a `Set` does not allow duplicates. All strings of this set are then processed in another stream that maps the strings to `int` values according to the length of each string. This results in the output `0124`, because sorting of values is done before printing.

In (b), another set of `Integer` objects is constructed based on the same values. However, in this case all `null` elements and empty strings are converted to the `int` value of `0`, and then removed from the stream by the `filter()` method. All values in the result set are then processed in another stream that sorts the elements and prints the output `124`.

(c) applies similar logic to that in (b), except that it uses a collector that assembles the values in a `List` rather than a `Set`. Duplicate elements are allowed in lists, resulting in the output `1224`.

(d) is similar to (c), but it applies the `distinct()` operation to the stream elements, removing any duplicates, and resulting in the output `124`.

17 Date and Time

17.1 (b)

The first `LocalDate` object represents the date January 31, 2021, thus representing 31 days since the start of the year. The second `LocalDate` object is the result of adding exactly one month to the first `LocalDate` object. Since 2021 is not a leap year, it represents the date February 28, 2021, which is 59 days from the start of the year. The third `LocalDate` object is the result of subtracting one month from the second `LocalDate` object, resulting in the date January 28, 2021.

17.2 (a)

A `LocalDate` object is initially constructed to represent the date January 1, 2021. A new `LocalDate` object is then constructed based on this date, by first changing the day of the month to be 31, which is the last day of this month. Next, the month in this date is changed to February. It is important to remember that February in 2021 has only 28 days, so the resulting `LocalDate` object would have to represent the last day of February.

17.3 (d)

The `LocalDateTime` denoted by `d1` represents
`2021-04-01T00:00`.
The method `toInstant()` converts `d1` by applying zone offset +18:00—that is, 18 hours ahead of the time at UTC.
To convert `d1` to an `Instant` denoted by `i1` at zero UTC offset, we must subtract 18 hours, resulting in the instant
`2021-03-31T06:00:00Z`.
The `ofInstant()` method converts `i1` to `LocalDate`, but no offset adjustment is necessary to the date represented in `i1` as it represents a point in time on the UTC timeline.

17.4 (c)

Two `ZonedDateTime` objects are constructed exactly one hour apart. However, two new zoned date-time objects are create from these two, and the duration between them is calculated. These new zoned data-time objects are an extra hour apart because one subtracts and the other one adds 30 minutes, thus increasing the time difference to two hours between 23:30CET and 00:30GMT. Another way to view this is to convert the time in one time zone (23:30CET) to the other time zone (22:30GMT). The difference between 22:30GMT and 00:30GMT is two hours.

17.5 (d)

Both `Instant` and `LocalTime` can express values with nanosecond precision. The `between()` method of the `Duration` can be used to calculate the time difference between two temporal objects—that is, between objects of the classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `Instant`. The `between()` method of the `Period` can be used to calculate the date-based amount of time between two `LocalDate` objects.

17.6 *(c)*

A `LocalDateTime` object is created that represents `2021-04-01T08:15`. Thirty minutes are subtracted from this date-time object, returning a new date-time object. The day of the month is set to 12 for the resulting date-time object. However, the reference value of the final object is not assigned to any reference. `LocalDateTime` objects are immutable, thus the date-time object denoted by `dt` is never modified.

17.7 *(e)*

Unlike `Period`, when `Duration` is applied to a `ZonedDateTime` object, it disregards daylight savings.

A `Period` is a date-based amount of time (in terms of years, months, and days), and therefore cannot express an amount of time smaller than one day. A `Period` of one hour cannot be created.

Unlike `ZonedDateTime` objects, `LocalTime` and `LocalDateTime` objects have no time zone, so they do not take into consideration daylight savings.

A `Period` of one day may or may not be treated the same as a `Duration` of 24 hours, when using these objects with `ZonedDateTime` because of the differences in the handling of daylight savings.

Finally, both `Period` and `Duration` can express positive and negative amounts of time.

17.8 *(e)*

The `plus()` method of the `LocalDate` class returns a `LocalDate`. The `plus()` method of the `LocalDateTime` class returns a `LocalDateTime`.

17.9 *(d)*

The `plus()` method of the `LocalDate` class can accept a `Duration` object, but in this scenario it will throw an exception at runtime. The reason for this is that `Duration` is expressed in seconds and nanoseconds, which cannot be applied to a `LocalDate` object.

17.10 (a) and (e)

In order to compute the desired result, a time of 30 minutes and two days should be added to the given `LocalDate` object.
(a) combines a `LocalTime` object with a value of 30 minutes to the `LocalDate` object using the `atTime()` method, returning a `LocalDateTime` object. Then a duration of 48 hours is added to this `LocalDateTime` object.
(b) attempts to add a `Duration` of 48 hours to the `LocalDate` object, which will result in an `UnsupportedTemporalTypeException`. The reason for this is that `Duration` is expressed in seconds and nanoseconds, which cannot be applied to a `LocalDate` object.
(c) attempts to create a `LocalTime` object, with an amount of time greater than 23 hours, which is invalid for a `LocalTime` object, and will result in a `DateTimeException`.
(d) attempts to create a `LocalTime` object with a negative number of hours, which is also invalid, and would result in a `DateTimeException`.
(e) combines a value of 30 minutes to the `LocalDate` object using the `atTime()` method and returning a `LocalDateTime` object. Then a duration of 48 hours is added to this `LocalDateTime` object. The `atTime()` method is an overloaded method.

17.11 (c)

Five `LocalDate` objects are created in this example. The `atTime()` method creates a `LocalDateTime` object. The other five methods create a new `LocalDate` object.

17.12 (d)

The method `between()` calculates the amount of time that has elapsed between a `LocalTime` object and a `LocalDateTime` object. A `LocalTime` object is derived from the second argument, which is a `LocalDateTime` object. The result would have been a runtime exception if the two arguments had been interchanged: We cannot derive a `LocalDateTime` object from a `LocalTime` object.
The value of the first argument of the `between()` method represents a time that is after the time represented by the derived `LocalTime` object. Therefore, the resulting `Duration` object will have negative values. The time difference between

17:30 and 15:15 is two hours and 15 minutes.

17.13 *(d)*

The required `LocalDateTime` object is exactly 25 hours (one day and one hour) ahead of the initial `LocalDateTime` object.
(a) adds one hour to the initial `LocalDateTime` object and changes the date to the next day.
(b) adds one day to the initial `LocalDateTime` object and changes the time by one hour.
(c) adds two days to the initial `LocalDateTime` object and subtracts 23 hours, which results in the required 25 hours being added to the initial `LocalDateTime` object.
(d) adds two days to the initial `LocalDateTime` object and subtracts 16 hours and 15 minutes, which results in the `LocalDateTime` object having the value `2021-04-03T23:15`.
(e) and (f) both adds a `Duration` of 25 hours (60 * 25 minutes) to the initial `Local-DateTime` object.

18 Localization

18.1 *(a)*

For the French locale, the resource bundle for the default locale (US) is loaded, as there is no resource bundle file named `MyResources_fr_FR.properties`. The values of the keys in this resource bundle are printed. The key `"farewell"` has duplicates. The last value (`"Bye!"`) specified for this key is returned.

18.2 *(b)*

The resource bundles loaded for `Locale.ENGLISH` are:

```
MyResources_en.properties
MyResources.properties
```

The method `getString()` returns the value `"Have a good one!"` for the key `"farewell"` in the resource bundle file `MyResources_en.properties`.

18.3 *(b)*

The code prints all available key–value pairs in the resource bundle for the English locale. The resource bundles loaded for the English locale are:

```
resources_en.properties
resources.properties (parent)
```

The key set contains the keys from both bundles: `k1` and `k2`. The key `k1` is found in the `resources_en.properties` bundle with the value `c`, and the key `k2` is found in the `resources.properties` bundle with the value `b`. Only if a key is not found in the current resource bundle, will it be looked up in its parent bundle, and so on. The `resource_en_GB.properties` bundle is not loaded by this code.

18.4 *(b)*

The code sets the default locale to the Russian locale, but then prints all available key–value pairs in the resource bundle for the English locale. The resource bundles loaded for the English locale are:

```
resources_en.properties
resources.properties (parent)
```

As appropriate bundles were found for the English locale, the default locale Russian bundle is not loaded.

The key set contains the keys from both bundles: `k1` and `k2`. The key `k1` is found in the `resources_en.properties` bundle with the value `c`, and the key `k2` is found in the `resources.properties` bundle with the value `b`. Only if a key is not found in the current resource bundle, will it be looked up in its parent bundle, and so on.

18.5 *(a), (b), (c), (d), and (e)*

The patterns produce the following output: Pattern Output

Click here to view code image

```
           Pattern      Output

    (a)   .00       |.46|
    (b)   .##       |.46|
    (c)   .0#       |.46|
    (d)  #.00       |.46|
    (e)  #.0#       |.46|
    (f)  #.##       |0.46|
    (g)   .#0      Throws java.lang.IllegalArgumentException.
```

18.6 *(c)*

Notice that the code sets the number of decimal digits to two. First, the rounding mode is set to `HALF_UP`, which would round a double value of 9876.54321 to 9876.54. Then the same number is formatted again, but this time with the rounding mode set to `HALF_DOWN`, which would actually produce the same result. Because the third digit after the decimal point is 3 and not 5, it has no effect on the rounding by the `HALP_UP` or `HALF_DOWN` rounding mode.

18.7 *(d)*

Notice that the code sets the number of decimal digits to two. The value is a `Big-Decimal` and therefore is represented with a higher precision than it would be in a `double` value. The third digit after the decimal point is 5; therefore, the discarded fraction is 0.5. In this case, `HALF_DOWN` mode rounds up if the discarded fraction is > 0.5, which it is not, resulting in the formatted value `$9,876.54`. `HALF_UP` mode rounds up if the discarded fraction is > = 0.5, resulting in the formatted value `$9,876.55`.

18.8 *(d)*

The default short date format for the British locale is `dd/MM/YYYY`. However, this is not relevant in this case, because a `DateTimeFormatter` is used to format a `LocalDate` object, which has no time part, and thus results in an `UnsupportedTemporalType-Exception`.

18.9 *(h)*

A `DateTimeFormatter` object is configured to use a pattern, where `d` stand for the day of the month, `a` for an am/pm marker and `y` for a year. It is configured to use the UK locale. Notice that the time value in the `LocalDateTime` object is 14:30, which makes it pm.

18.10 *(e)*

In the code, the default locale is initially set to UK (British). A `DateTimeFormatter` is created to format a date according to the `MEDIUM` style format (`MMM d, yyyy`). This formatter formats the date to the string `"Apr 1, 2021"`. The reference `s1` denotes this string.

Next, the default locale is set to France. However, the `DateTimeFormatter` is immutable, and the date is formatted to the string `"Apr 1, 2021"`. The reference `s2` denotes this string.

Lastly, the `localizedBy()` method returns a new `DateTimeFormatter` object with the US locale, but it is not assigned to any reference. The date is again formatted by the previous `DateTimeFormatter` to the string `"Apr 1, 2021"`. The reference `s3` denotes this string. This means that only the condition in the first `if` statement is true.

18.11 *(d)*

`NumberFormat` interprets a floating-point number as a percentage, considering 1.0 to be equal to 100%. The default percentage format object rounds the value to two decimal digits, so it would round the double value `0.987654321` to `0.99`. The value `0.99` represents 99%.

18.12 *(a)*

The code creates a `LocalDateTime` object (`date1`) first and adds a London time zone to it to create a `ZonedDateTime` object (`date2`).

Next, two `DateTimeFormatter` objects, `df1` and `df2`, are created based on the same pattern `"hz"`, and the time zones for London (GMT) and Paris (CET) are associated with them, respectively. The `"hz"` pattern stands for hour and time zone. Both `date1` and `date2` are formatted by each `DateTimeFormatter`.

The `DateTimeFormatter df1` will format the hour in `date1`, but will supply the time zone, as this date object has no time zone, creating the string `"1GMT"`. It will format `date2` as `"1GMT"`, as the `ZonedDateTime date2` and the `DateTimeFormatter df1` have the London time zone.

The `DateTimeFormatter df2` will format the hour in `date1`, but will supply the time zone, as a date object has no time zone, creating the string `"1CET"`. It will format `date2` as `"2CET"`, as the `ZonedDateTime date2` and the `DateTimeFormatter df2` have different time zones. Therefore, the London time (`1h`) in `date2` is interpreted as Paris time (`2h`) by the `DateTimeFormatter`.

18.13 *(d)*

Values in single quotes are treated as verbatim text, rather than format elements. Only two values are actually supplied for the message pattern, so the format element with index 3 receives no value and will thus be formatted as text.

18.14 *(b)*

During the format operation, elements of the `values` array are interpreted by the `MessageFormat` and `ChoiceFormat` objects to determine the limit value that affects the selection of the appropriate choice format and the value that should be substituted into a message pattern. In this code, the value `4` in `values[0]` is the limit value, and the value `5` in `values[1]` is the value to format, resulting in the choice pattern `"{1}th"` at `formats[4]` to be chosen to format the value `5`. The formatted result is `"5th"`.

18.15 (b)

The limit values are given by the `limits` array. Note that the limit values are not ordered.

Click here to view code image

```
limits[0]    limits[1]    limits[2]
   0            -1            1
```

The corresponding choice formats are given by the `formats` array:

Click here to view code image

```
formats[0]   formats[1]   formats[2]
  "zero"     "negative"   "positive"
```

The supplied value `0.9` is greater than 0, and of course also greater than –1, but less than 1. It satisfies the following relation, determining index 1 in the `limits` array to choose the choice format.

Click here to view code image

```
limits[1] <= 0.9 < limits[2]
  -1      <= 0.9 <    1
```

Index 1 in the `formats` array determines the choice format to be the string `"negative"`.
The value `0.9` results in the choice format at index 1 in the `formats` array to be chosen.

18.16 (a)

The default locale is set to the US locale and a `DateTimeFormatter` object is created to format dates in the `MEDIUM` style format (`MMM d, yyyy`).
At (1), a string is parsed using the `dtf` formatter, according to the `MEDIUM` date format style and the default locale (in this case, US). The string is specified in the `MEDIUM` format style for the US locale.
At (2), a string is parsed to a date. As no formatter is specified, the string is expected to be in the ISO format, which it is.
At (3), the `LocalDate d` is formatted using the `dtf` formatter that uses the `MEDIUM` format style and the default locale (in

this case, US).

18.17 *(c)*

The default locale is set to the US locale and a `DateTimeFormatter` object is created to format dates in the `SHORT` style format (`M-d-yy`).
At (1), a string is parsed to a date. As no formatter is used, the string is expected to be in the ISO format, which it is.
At (2), a string is parsed using the `dtf` formatter, according to the `SHORT` date format style and the default locale (in this case, US). The string is specified in the `SHORT` format style for the US locale.
Finally, this date is printed using the default ISO format.

18.18 *(d)*

The default locale can be defined explicitly; otherwise, it is the platform locale that is supplied by the runtime environment. The default locale is not necessarily the US locale. The default format for `LocalDate` objects is `ISO_DATE`.

19 Java Module System

19.1 *(b)*

Code in the module `ui` can access public types defined in the packages `store.fron-tend` and `store.backend` via direct dependency, but also public types defined in the package `product.data` via transitive dependency. However, module `ui` does not require module `customer`, thus it cannot access public types from the `customer.data` package. This means that (a) is false, but (b) is true.
Code in the module `customer` cannot access public types from the `product.data` package, despite the presence of transitive dependency via the modules `ui` and `store`. This is because module `customer` does not have a direct dependency on module `store`, which has a transitive dependency on module `product`. This means that (c) and (d) are both false.
Code in the module `product` cannot access public types from the `customer.data` package because of the absence of a dependency between these modules. This means (e) is false.

19.2 *(e)*

Only public types in the exported packages of a module are accessible to code in modules that require this module.

19.3 (e)

The `java.se` module is at the root of the module graph, as it depends on the highest number of modules in the graph. The `java.base` module is at the bottom of the module graph, and does not depend on any module. The `java.logging` module depends on the `java.base` module.

19.4 (c) and (f)

Only public types in the exported package `animals.primates` are accessible to code in module `zoo`.

19.5 (a) and (d)

Module `music` should declare a `requires` directive that should specify module `production`. Also, module `production` should export package `production.company`.

19.6 (c) and (d)

An automatic module is a plain JAR that is loaded from the module path. Plain JARs loaded from the class path are included in the unnamed module.

19.7 (a)

In (a), code in module `store` can access types defined in package `product.data` because this package is exported by module `product`, which module `store` actually requires. Code in module `store` can access types defined in package `marketing.offers` by reflection as this package is opened by module `marketing`.

(b) is incorrect because code in module `marketing` cannot access types defined in packages `product.data` and `product.pricing` because it does not require module `product`.

(c) is incorrect because code in module `marketing` cannot access types defined in package `product.data` as it does not require module `product`. However, code in module `marketing` can access types defined in package `store.frontend`, as module `marketing` requires module `store`.

(d) is incorrect because code in module `store` can access types defined in package `product.data`, but not in package `product.pricing`, as this package is only exported to module `marketing`.

(e) is incorrect because code in module `product` cannot access types defined in package `store.frontend` as module `product` does not have a dependency on module `store`. However, code in module `product` can access types in the open package `marketing.offers`.

19.8 *(b)*

The `requires` directive specifies module names, not package names, which disqualifies (a) and (d). In (c), module `music` does not depend on module `artist`, and therefore cannot access types in package `artist.recoding`. This leaves (b), which works because module `music` has a direct dependency on module `production`, and also has a dependency on module `artist` via the `requires-transitive` directive in module `production`.

19.9 *(b)*

Service consumer module `player` does not need to declare any dependency on service provider module `brass`. Service consumer module `player` needs to declare a dependency on service module `music`, and specify which abstract type (in this case `music.sound.Instrument`) defines the service.

19.10 *(b) and (e)*

The `jlink` tool creates platform-specific runtime images that can be deployed. Runtime images contain application code, as well as the necessary JDK modules and tools, among other artifacts. However, no installation of a separate JVM is required to run the application.

19.11 *(a) and (b)*

Module `music` requires module `instrument`, which in turn requires module `music`. This results in a cyclic dependency, and thus is illegal and would cause these module declarations not to compile. Both modules `music` and `artist` export the same package `preferences.style`, which implies that this package is a split package, which is illegal and would cause this module declaration not to compile. It is allowed to declare the `opens` directive in a module declaration. It is also allowed to declare a qualified exports directive even if the specified module does not require this module.

19.12 *(e)*

Both modular and plain JARs can be used in the context of the class path as well as the module path. A listing of modules using the `--list-modules` of the `java` tool includes all observable modules, but not the unnamed module, as it has no name. The name of an automatic module is derived from the JAR file name, unless it is specified in the `MANIFEST.MF` file. There is only one unnamed module, and it obviously does not have a name, which makes the JAR file name irrelevant in this case.

19.13 (c) and (d)

An automatic module implicitly requires all other modules. An explicit module cannot access code in the unnamed module using the `requires` directive, since the unnamed module has no name. If an explicit module needs to access code in an automatic module, it must declare a `requires` directive to specify this automatic module.

20 Java I/O: Part I

20.1 (d)

The `read()` method will return `-1` when the end of the stream has been reached. Normally an unsigned 8-bit `int` value is returned (range from `0` to `255`). I/O errors result in an `IOException` being thrown.

20.2 (d)

The `print()` methods of the `PrintWriter` do not throw an `IOException` when the end of the file is reached, but instead sets an error status that can be checked.

20.3 (d)

The `read()` method of an `InputStreamReader` returns `-1` when the end of the stream is reached.

20.4 (b)

The `readLine()` method of a `BufferedReader` returns `null` when the end of the file is reached.

20.5 (c)

An `ObjectOutputStream` can write both objects and Java primitive types, as it implements the `ObjectInput` and the `DataInput` interfaces. The serialization mechanism will follow references in objects and write the complete object graph.

20.6 (a), (b), (d), and (i)

Static fields and `transient` instance fields are not serialized—they are treated the same way when it comes to serialization. The accessibility modifier `private` does not determine whether an instance field should be serialized or not. `Serializable` is a marker interface. Subclass objects are serializable if the superclass is serializable. The modifier `final` of a class does not determine whether the class is serializable or not.

20.7 *(b)*

(a) is incorrect because there is no requirement that all versions of a serializable class must provide a declaration of a `serialVersionUID`.

In (b), there is no guarantee that a streamed object based on one version can be deserialized based on the other, even if the two versions of the class have the same `serialVersionUID`. It depends on whether the changes in the class versions are compatible with deserialization. For example, changing the type of a field is an incompatible change.

(c) is incorrect because `serialVersionUID` in two unrelated serializable classes is also unrelated and need not be unique.

(d) is incorrect because there is no requirement that the `serialVersionUID` of a serializable class must be incremented every time a new version of the class is created.

(e) is incorrect because any class can declare a `static final` field of type `long` having the name `serialVersionUID`, but it has meaning for serialization only in a serializable class.

20.8 *(e)*

During deserialization, the zero-argument constructor of the superclass `Person` is called because this superclass is not `Serializable`.

20.9 *(c)*

If the superclass is `Serializable`, then the subclass is also `Serializable`—resulting in the printout in (c).

20.10 *(e)*

Note that only `GraduateStudent` is `Serializable`. The field `name` in the `Person` class is `transient`. During serialization of a `GraduateStudent` object, the fields `year` and `studNum` are included as part of the serialization process, but not the field `name`. During deserialization, the private method `readObject()` in the `GraduateStudent` class is called. This method first deserializes the `GraduateStudent` object calling the no-argument constructor in the superclasses, but then initializes the fields with new values. Without the private `readObject()` method, the output would be as in (d).

20.11 *(a)*

Constructors for `Product` and `Food` are triggered when a new

`Food` object is created. These constructors print `"product food "`. After that, the `product` object is serialized. `Product` is a superclass of `Food` and is marked as `Serializable`, which implies that all of its subclasses, such as `Food`, are also serializable. Values of `Product` name and `Food` calories are included in the serialization of the `product` object.
No constructors are triggered during deserialization, thus `Product` and `Food` constructors do not print any values. No errors are triggered during deserialization, and values of `Product` name and `Food` calories are restored and printed.

20.12 *(c)*

A buffer (a `char` array of length 4) is used to read and write characters to files. In the `while` loop, this buffer is filled with characters from the `test1.txt` file and written to the `text2.txt` file. On the first iteration, the characters `a`, `b`, `c`, and `d` are read from the `text1.txt file`, filling the buffer to full capacity, and then written to the t`est2.txt` file. On the second iteration, the remaining characters `e`, `f`, and `g` are read from the `test1.txt` file into the buffer. These characters are read into the first three elements of the buffer. The fourth element in the buffer still contains the character `d` from the previous read operation. The buffer contains the characters `e`, `f`, `g`, and `d`, which are written to the `text2.txt` file. The next read operation returns `-1` since the end of the file has been reached in the `text1.txt` file, thereby terminating the loop.

20.13 *(c)*

Both fields `numberOfTracks` and `currentTrack` are not included when an `Album` object is serialized, as the field `numberOfTracks` is `static` and the instance field `currentTrack` is `transient`. Only the instance field `title` (having the value `"Songs"`) is included in the serialization of the `Album` object.
The `readObject()` method of the `Album` class is not `private`, but `public`, and is never called during deserialization to change the state of the `Album` object created at deserialization. The `Album` object created at deserialization is initialized with the instance field `title` having the value `"Songs"` and the `transient` field `currentTrack` initialized to the default `int` value `0`.
Deserialization requires definition of the class, thus an `Album` object created at deserialization can access the `static` field `numberOfTracks` in its class that has the value `5`.

21 Java I/O: Part II

21.1 (b) and (c)

Compiling and running the program results in the following output:

Click here to view code image

```
/wrk/./document/../book/../chapter1
/wrk/chapter1
chapter1
./document/../book/../chapter1
./document/../book/..
```

Note that only the `Path.toRealPath()` method requires that the file exists; otherwise, it throws a `java.io.IOException`.

21.2 (c)

Compiling and running the program results in the following output:

```
./wrk/src

./wrk
./wrk/src
./wrk/src/readme.txt

./wrk
./wrk/src
./wrk/src/readme.txt
```

The `Files.list()` method creates a stream based on the *immediate* entries of the directory path passed as a parameter. The `Files.walk()` method traverses depth-first every entry in the hierarchy of the directory passed as a parameter. The `Files.find()` method will find every entry in the hierarchy of the directory passed as a parameter, since the matcher argument is always `true` and will traverse to depth 2 (i.e., equal to the actual depth of the directory).

21.3 (a)

There are three absolute `Path` objects (starting at the root (/) of the file system) constructed in this example: `Path earth` is defined as `"/planets/earth"`.

`Path moonOrbit` is defined as `"/planets/earth/moon/orbit.param"`—that is, as a child of `Path earth`.

`Path mars` is defined as `"/planets/mars"`—that is, as a sibling of `Path earth`.

There is one relative path:

`Path fromMarsToMoon` is defined as `"../earth/moon/orbit.param "`—that is, as a relative path between `Path mars` and `Path moonOrbit`.
These `Path` objects do not have to actually exist in the file system, so long as a program makes no attempt to validate or access these paths. Thus no runtime exception will be thrown.

21.4 *(b)*

First, a `Path` object is created with the relative path `"./mars/../earth"`, that has four name elements. Next, this `Path` is normalized, resulting in a `Path` object with the path string `"earth"` that has one name element. Then it is converted to the absolute path `"/planets/earth"` which has two name elements. If this path had not have been normalized, then the absolute path would be `"/planets/./mars/../earth"`, which has five name elements. Whether the paths exists in the file system is irrelevant, since this program makes no attempts to actually validate or access any of these paths.

21.5 *(d)*

The `list()` method of the `Files` class creates a stream of `Path` objects denoting the immediate entries in the given directory. Unlike the method `walk()`, the `list()` method does not traverse the contents of the subdirectories. The stream created by the `list()` method will include the `Path` objects that denote the following paths:

```
/test/a.txt
/test/c
/test/e.txt
/test/f.txt
```

A filter is applied to this stream of `Path` objects, which uses the method `getFileName()` of the `Path` interface to return the last name element of the `Path`. The filter will discard any entry whose file name does not end with `"txt"`. This leaves only the following paths in the stream:

```
/test/a.txt
/test/e.txt
/test/f.txt
```

These paths are then printed to the console.

21.6 *(g)*

The `walk()` method of the `Files` class creates a stream of `Path` objects that denote all entries in the given directory, including its subdirectories, by traversing the directory hierarchy depth-first. The stream will include `Path` objects that denote the following paths:

```
/test/a.txt
/test/a.txt/b.txt
/test/c
/test/c/d.txt
/test/e.txt
/test/f.txt
```

A map operation is applied to this stream of `Path` objects, converting it to a stream of `String` objects, where each `String` represents the last name element of the path— that is, the file name:

```
a.txt
b.txt
c
d.txt
e.txt
f.txt
```

A filter is applied to this stream of `String` objects which excludes strings that do not end in the file extension `"txt"`. This leaves only the following paths in the stream (notice that these strings are sorted):

```
a.txt
b.txt
d.txt
e.txt
f.txt
```

These `String` objects are then printed to the console.

21.7 *(b)*

The method `Files.createDirectories()` does not throw an exception when trying to create a directory that already exists. However, the method `Files.createDirectory()` does. Method `Files.delete()` does throw an exception when trying to delete a nonexistent directory.
Method `Files.move()` does throw an exception when moving a non-empty directory, but only if it actually needs to move all files within this directory to another file system. Moving a directory within the same file system does not actually perform any move operations for the directory. It only changes the path of the directory. Method `Files.exists()` returns a `boolean` value to indicate the existence or

nonexistence of a path.

21.8 (c)

Two `Path` objects are initialized. However, both of these `Path` objects reference the same file. This is because `p1.getName(1)` returns a relative path to directory `joe`, which is the second component in this path, considering that the first component is directory `users` with index 0. This path is then used to resolve another relative path `test/a.jpg`, which results in the path `joe/test/a.jpg`. And finally, the path `/users` is used as a root directory to resolve this path, resulting in the path `/users/joe/test/a.jpg`, which is identical to the path referenced by `p1`.

Next, an attempt is made to move the entry at path `p1` to path `p2`, where both denote the same directory entry. In this scenario, `Files.move()` method performs no action because it can detect that both the source and the destination path are the same.

21.9 (c)

In this example, the `p1` reference represents an absolute path, and the `p2` reference represents a relative path. Keep in mind that an absolute path starts from the root of a file system, in this case designated by the slash character (`/`).

The purpose of the method `resolve()` is to construct a path where a relative path is appended to another relative path, or to an absolute path. `p1.resolve(p2)` results in the relative path `store` being appended to the absolute path `/test`. As it is not possible to append an absolute path to another path, `p2.resolve(p1)` returns the absolute path `/test` denoted by the reference `p2`.

21.10 (d)

This question assumes the existence of the destination file, yet this code example does not specify the replace-existing file copy option:

Click here to view code image

```
Files.copy(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

Therefore, the code will throw a `java.nio.file.FileAlreadyExistsException`.

21.11 (d)

The code reads lines of text from a file as a stream of strings.

A filter operation discards lines that do not start with `<`. Then each line in the stream is mapped by the map operation to a `>`. The reduction operation is applied to concatenate each `>`. The resulting string `">>>>>"` is printed as there are five lines that are mapped to `>`.

21.12 *(d)*

Only permissions explicitly added to the set are applied, all other permissions are removed. In order to be able to access files inside a directory, the directory needs to have execute permission. However, the permissions for the `/test/data` directory are changed to read-only in the code. Therefore, an attempt to access the `info.txt` file in this directory by the `walk()` method will throw an `AccessDeniedException`, that will terminate the stream processing.

21.13 *(d)*

The thing to note is that a `PosixFileAttributeView` can be used to set permissions for the file that is associated with it. However, the `PosixFileAttributes` object obtained from the `PosixFileAttributeView` will only reflect the file attribute values at the time it was obtained from the view. It is not updated automatically when the file attribute values change in the file. A new `PosixFileAttributes` object must be obtained from the view to reflect any changes in the file attribute values.

The program first removes all permissions from the file. A `PosixFileAttributeView` is created on the file, and the `PosixFileAttributes` object associated with the view is obtained. This `PosixFileAttributes` object is used in the rest of the program, and it will always reflect that the file has no permissions, regardless of any permissions set in the file through the view.

Note also that removing an element (`OWNER_WRITE` permission) from an empty set does not throw an exception. The permissions of the file are changed as follows, but the changes are not reflected by the `PosixFileAttributes` object:

```
---------
r---w-r--
-w-------
```

21.14 *(a)*

First, this program converts a `URI` object to a `Path` object. Path `p1` references the same file as path `p2`. No action is taken by the `copy()` method when the source file and the destination file are the same. Lastly, a `Path` object is converted to a legacy `File` object. This program runs successfully and produces no output.

22 Concurrency: Part I

22.1 *(e)*

The program will compile without errors, and will simply terminate without any output when run. Two thread objects will be created, but they will never be started. The `start()` method must be called on the thread objects to make the threads execute the `run()` method asynchronously.

22.2 *(d)*

Note that calling the `run()` method on a `Thread` object does not start a thread. In the statement:

Click here to view code image

```
new Thread(new R1(),"|R1a|").run();
```

the `run()` method of the `Thread` class will invoke the `run()` method of the `Runnable` object (`R1`) that is passed as an argument in the constructor call. In other words, the `run()` method of the `R1` class is executed in the `R2` thread—that is, the thread that called the `run()` method of the `Thread` class and whose name will be printed. However, the statement:

Click here to view code image

```
new Thread(new R1(),"|R1b|").start();
```

starts the `|R1b|` thread, and the `run()` method of the `Thread` class will invoke the `run()` method of the `Runnable` object (`R1`) that is passed as an argument in the constructor call, but it is executed by the `|R1b|` thread whose name will be printed. The last statement in the `run()` method of the `R2` class is executed by the `|R2|` thread whose name will be printed.

22.3 *(c)*

Note that the complete signature of the `run()` method does not specify a `throws` clause, meaning it does not throw any *checked* exceptions. However, a method can always be implemented with a `throws` clause that specifies *unchecked* exceptions, as in the case of the `run()` method.

22.4 (a) and (e)

Because the exact behavior of the thread scheduler is undefined, the text `A`, `B`, and `End` can be printed in any order. The thread printing `B` is a daemon thread, which means that the program may terminate before the thread manages to print the letter.

22.5 (a) and (e)

The lock is also released when an uncaught exception occurs in the statement.

22.6 (c) and (d)

First note that a call to `sleep()` does not release the lock on the `Smiley.class` object once a thread has acquired this lock. Even if a thread sleeps, it does not release any locks it might possess.
(a) does not work, as `run()` is not called directly by the client code.
(b) does not work, as the infinite `while` loop becomes the critical region and the lock will never be released. Once a thread has the lock, other threads cannot participate in printing smileys.
(c) works, as the lock will be released between each iteration, giving other threads the chance to acquire the lock and print smileys.
(d) works for the same reason as (c), since the three print statements will be executed as one atomic operation.
(e) may not work, as the three print statements may not be executed as one atomic operation, since the lock will be released after each print statement. Synchronizing on `this` does not help, as the printout from each of the three print statements executed by each thread can be interspersed.

22.7 (d)

A thread terminates when the execution of the `run()` method ends. The call to the `start()` method is asynchronous—that is, it returns immediately, and it moves the thread to the *`READY`* substate. Calling the `sleep()` or `wait()` method will block the thread.

22.8 (b) and (d)

The nested `createThread()` call is evaluated first, and will print `23` as the first number. The last number the main thread prints is `14`. After the main thread ends, the thread created by the nested `createThread()` completes its `join()` call and prints `22`. After this thread ends, the thread created by the outer `createThread()` call completes its `join()` call and prints the number `12` before the program terminates.

22.9 *(e)*

The exact behavior of the scheduler is not defined. There is no guarantee that a call to the `yield()` method will grant other threads use of the CPU.

22.10 *(b)*

The `final` method `notify()` is defined in the `Object` class.

22.11 *(c)*

An `IllegalMonitorStateException` will be thrown if the `wait()` method is called and the current thread does not hold the lock of the object.

22.12 *(d)*

Since the two methods `emptying()` and `filling()` are `synchronized`, only one operation at a time can take place on the tank that is a shared resource between the two threads. The method `emptying()` waits to empty the tank if it is already empty (i.e., `isEmpty` is `true`). When the tank becomes full (i.e., `isEmpty` becomes `false`), it empties the tank and sets the condition that the tank is empty (i.e., `isEmpty` is `true`).
The method `filling()` waits to fill the tank if it is already full (i.e., `isEmpty` is `false`). When the tank becomes empty (i.e., `isEmpty` becomes `true`), it fills the tank and sets the condition that the tank is full (i.e., `isEmpty` is `false`).
Since the tank is empty to start with (i.e., `isEmpty` is `true`), it will be filled first. Once started, the program will continue to print the string `"filling"` followed by the string `"emptying"`. Note that the `while` loop in the `pause()` method must always check against the field `isEmpty`.

23 Concurrency: Part II

23.1 *(c)*

A single thread executor service does not allow scheduling of tasks, but a scheduled executor service allows a task to be scheduled with a specified delay, and also allows a task to be executed periodically. The work stealing mechanism is specific to the work stealing thread pool, and the work stealing thread pool is designed to maintain enough threads to support a given level of parallelism.

23.2 *(b)*

The `shutdown()` method of the executor service initiates the shutdown of the executor service, allowing currently running tasks to continue, but preventing new tasks from being submitted. It does not wait for the termination of currently running tasks. The `awaitTermination()` method can throw an `InterruptedException`. The `shutdownNow()` method also initiates the shutdown of the executor service, but it cancels all running tasks and returns.

23.3 *(b)*

The read lock does not prevent other operations from reading data. Methods that acquire read and write locks can throw an `InterruptedException`. The write lock is designed to allow only a single exclusive write operation on the data, preventing other read and write operations to be performed, thus preserving memory consistency and preventing data corruption.

23.4 *(b)*

The contents of `list1` are created by spawning a number of threads, concurrently writing data to the copy-on-write list. There is no guarantee that these threads would actually complete copying all elements from the initial list by the time `list1` is printed.

The contents of `list2` are created by processing elements from the initial list using a parallel stream with a list collector reduction operation combining processed data into a single list. The `collect()` terminal operation ensures that the stream is exhausted and the collector ensures that all data is assembled into the list. When printed, `list2` has the same contents as the initial list.

The contents of `list3` are created by processing elements from the initial list using a parallel stream, but manually adding elements into `list3`. This does not guarantee the consistency of `list3` because of potential contention between threads trying to access `list3`, and is likely to corrupt data.

23.5 *(d)*

Synchronized collections provide blocking (synchronized) methods that modify collection content. However, synchronized collections do not provide a synchronized iterator. It is the programmer's responsibility to implement synchronized iteration behavior for such collections. Copy-on-write collections achieve concurrency by creating a copy of a collection for each thread that tries to modify the collection, and then automatically merging these copies without any need to implement thread synchronization. Immutable collections are read only, and thus are automatically considered to be memory safe, without any need for synchronization.

23.6 *(d)*

Atomic variables are designed to be thread-safe without the use of synchronization and intrinsic locking. Methods provided for atomic variables do not throw an `InterruptedException`.

23.7 *(a) and (b)*

The Atomic API provides operations that guarantee object consistency. However, no specific order is enforced when a number of atomic operations are performed concurrently. In this example, a number of `incrementAndGet()` calls are executed on an `AtomicLong` object, resulting in the consistent increment of its value. This means that the last value in this example would always be `3`. The order of the `Future` objects is the same as that of the invoked tasks, but the stream iterating through the list of these `Future` objects can print the numbers 1, 2, and 3 in any order, as the order of the concurrent increment operations is unpredictable.

23.8 *(a)*

In this example, an attempt is made to upgrade a read lock to a write lock, which is not possible. It should be noted that a write lock can be downgraded to a read lock. Consider the following scenarios: Attempting to obtain the read lock using the `lock()` method after the write lock has been acquired is allowed:

```
writeLock.lock();
readLock.lock();
```

Attempting to obtain a write lock using the `lock()` method after the read lock has been acquired will not succeed:

```
readLock.lock();
writeLock.lock();
```

Attempting to obtain a write lock using the `tryLock()` method after the read lock has been acquired will return `false`—that is, the write lock is not acquired:

```
readLock.lock();
writeLock.tryLock();
```

In the `finally` block, the `isWriteLocked()` method checks whether the write lock has been acquired, but we have already established that this would not be the case. So only the `"Read lock acquired"` and `"The end"` messages will be printed in this scenario.

23.9 *(d)*

In this example, the variable `counter` is declared as `volatile`. A `volatile` variable guarantees visibility of write operations. It does not guarantee memory consistency when several concurrent threads attempt to modify this volatile variable with a non-atomic operation (`--`). There is a danger of interleaving of read and write operations on the variable by different threads; thus the results are unpredictable.

23.10 *(e)*

The submit() method does not throw an exception. It is possible to submit both a `Callable` (`() -> "acme"`) that returns a value and a `Runnable` (`() -> {}`) that does not. The `shutdown()` method does not throw an exception. An invocation of the `shutdown()` method initiates the shutdown of the executor service, but the two already submitted tasks are allowed to complete. However, the `shutdown()` method does not wait for the tasks to complete execution. Although the `get()` method can throw checked exceptions, no exceptions are thrown in this case. Invocation of the `get()` method on a `Future` blocks until the task represented by the `Future` completes execution. The first `get()` method call returns the result of executing the `Callable`, which in this case returns the string `"acme"`. Since a `Runnable` does not return a value, the second `get()` method call returns the `null` value to indicate normal completion of the task represented by the `Runnable`. The print statement does not throw an exception. It prints `"acme null"`, which is the concatenation of the results `"acme"` and the `null` value returned by the `get()` methods, respectively.

23.11 *(f) and (h)*

In the `for(;;)` loop, there is no guarantee that a task will actually be cancelled before it completes. Cancelled or not, each task is added to the `results` list. The `shutdown()` method initiates the shutdown of the executor service, allowing those tasks that are already running to complete execution.

The method `isDone()` only returns `true` if the task completed due to normal termination, an exception, or cancellation. If any task was still running, the `allMatch(r>r.isDone())` expression will return `false`, causing the letter `Z` to be printed.

If all tasks completed, then there could be some among them that were cancelled. An attempt to get the value of a `Future` whose task has been cancelled will result in an exception. In order to concatenate the values returned by the tasks in the `Future` objects, all cancelled tasks are filtered from the stream by calling the `isCancelled()` method. Since it is unpredictable which tasks were cancelled and which terminated normally, the output from the program may contain any of the letters `A`, `B`, `C`, `D`, or `E`.

24 Database Connectivity

24.1 *(c)*

When no rows are returned by the query, invoking the `next()` method simply returns `false` to indicate the absence of the next row in the result set. This is considered normal behavior and does not cause an exception, ruling out (a) and (b). The first row in the result set has index 1.

24.2 *(c)*

Programmers should ensure that result sets are closed first, then statements, and only then the connection. The closure order is important because unclosed result sets and statements can cause a memory leak on the database side. This has nothing to do with the transactional behavior of the program, and thus is not related to auto-commit mode.

24.3 *(b)*

In this code example, the marker parameter in the select statement is set to match rows that start with the string `"Where"`. The table contains exactly two rows that match the where clause, so these rows will be retrieved and printed by the program. When retrieving a column value, the column might not have a value—that is, it might be `null`. Testing a reference for `null` before using the reference avoids the `NullPointerException` at (3).

24.4 *(b)*

Notice that in this code example, the value `103` that is set for the `id` column in the query does not match any id in the rows in the table. Therefore, when executed, this query will not return any rows. This is not an error, so it would not cause any exceptions. Instead, the method `next()` will return `false`, as the result set is empty. As a result, the body of the `if` statement will not be executed and nothing will be printed.

24.5 *(b)*

First note that the auto-commit mode has been disabled for the connection. Next, this program sets the marker parameters and executes an update statement. However, when it sets the marker parameter for the select statement, it uses index 2, but there is only one marker parameter in this select statement, so the line of code `ps1.setInt(2, id)` will throw an exception. This will interrupt normal program execution and control will be transferred directly to the `catch` block. This means that the explicit commit statement will not be executed.
However, the `catch` block does not attempt to roll back this transaction, so once the exception is handled by the `catch` block, the program will resume its normal execution, which will correctly close the statements and the connection in the implicit `finally` block of the outer `try`-with-resources statement. Therefore, the database will not have any indication that it is supposed to perform a rollback, as no rollback is executed in the `catch` block. Its reaction to a normal connection closure is to commit any outstanding changes.

24.6 *(b)*

The resources will be closed in the following order: result set, statement, and connection.

24.7 *(b)*

Marker parameters in a prepared statement are set with the `setXXX()` methods, not with the `executeQuery()` method, which rules out (a). Each prepared statement represents a single SQL statement, which can be parameterized and executed multiple times, contradicts (c) and (d).

24.8 *(a) and (b)*

The SQL query in this example selects rows from the `questions` table, using a `where` clause that selects rows that do not have any value (`null`) for the `answer` column. Then the code iterates through the result set containing these rows and updates the answer column value to be `"no answer"`.
(c) is incorrect because the `SELECT` statement will return some, but not necessarily all, rows from the `questions` table.
(d) is incorrect because potential exceptions will be caught by the `catch` block, which does not invoke the `rollback()` method. Therefore, once an exception is caught, the program will resume normal execution and will correctly close the connection in the implicitly `finally` block of the outer `try-with-resources` statement. Databases assume that if a program has correctly closed its JDBC connection, then there is no reason not to commit any outstanding changes made within the context of this connection.

24.9 *(c)*

There are two problems with the code.
First, only one of the marker parameters is actually set before the statement is executed.
And second, an update SQL operation cannot be executed using the `execute-Query()` method.
Either one of these issues would cause the `executeQuery()` method to throw an exception.

24.10 *(a)*

Marker parameters in a prepared statement need not be set in any specific order, as long as they are all set before the statement is executed. In the given code, all marker parameters are set before the statement is executed, so the code executes normally.

24.11 *(b), (c), and (d)*

A prepared statement can be executed multiple times, making (b) a correct option. If a statement is a `SELECT` statement, then it can be executed using either the `execute()` or `executeQuery()` methods; otherwise, it can be executed using the `execute()` and `executeUpdate()` methods, making (c) and (d) correct options. Statements executed by the `executeQuery()` method and the `executeUpdate()` method are mutually exclusive—the former executes `SELECT` statements and the latter non-`SELECT` statements like `INSERT`, `UPDATE`, and `DELETE`. The `execute()` method can execute all statements.

24.12 (a)

The default navigation direction in a result set is forward only, meaning starting with the first row and successively proceeding to the last row each time the `next()` method is called. Forward-only navigation is supported by all databases. Other result set options, such as reflection of changes, scroll sensitivity, and cursor closure on commit, may or may not be supported by different databases.

24.13 (a)

The `relative(int rows)` method moves the cursor a specified number of rows in relation to the current row in the result set. The parameter value can be a positive or a negative `int` value.
Calling the method `relative(1)` moves the cursor forward by one row, which is the same as calling the method `next()`.
Calling the method `relative(0)` does not move the cursor from its current position. Calling the method `relative(-1)` moves the cursor backward by one row, which is the same as calling the method `previous()`.
Calling the method `absolute(1)` moves the cursor to the first row in the result set, which is the same as calling the method `first()`.
Calling the method `absolute(0)` moves the cursor to before the first row in the result set.
Calling the method `absolute(-1)` moves the cursor to the last row in the result set, which is the same as calling the method `last()`.

25 Annotations

25.1 (a) and (b)

Annotations are compiled into classes just like any other classes or interfaces. The purpose of an annotation is to provide metadata for program elements in the code (like Java classes, interfaces, methods, and variables). Annotations can be applied to other annotations, and can also be used as a type of an annotation element.

25.2 *(c)*

Annotations having the target `ElementType.TYPE` can be applied to classes, interfaces, enums, and other annotations. Annotations having the target `Element-Type.TYPE_PARAMETER` can be applied to type parameters in generic code. Annotations having the target `ElementType.FIELD` can be applied to constants, as constants are static fields. Annotations having the target `ElementType.METHOD` can be applied to only methods, but annotations having the target `ElementType.CONSTRUCTOR` can be applied to constructors.

25.3 *(a) and (c)*

The annotation in question is applied to the class, and requires no parameters to be supplied. This means that its target must be either default, or explicitly declared to be applicable to `ElementType.TYPE`. This would exclude (b) and (d).

25.4 *(b) and (g)*

The annotation type declaration defines a single element type of `int` array called `value`, and provides a default value for this element. This question asks to identify incorrect ways of applying this annotation. (a) and (f) supply a single value for this annotation element, in which case no `{}` are required to enclose the value, and it is not required to specify the element name `value` when only one value is specified. (c) is legal because the element name `value` does not have to be specified when it is the only element specified in the annotation type declaration. (d) and (e) are legal because the `value` element does not have to be set, since it has a default value, and parentheses `()` are optional when no value is specified. (b) and (g) are illegal because they are missing the block notation `{}` to enclose the list of values specified.

25.5 *(b)*

The annotation type `Test5Annotation` should be defined to be applicable to at least the targets of `TYPE`, `TYPE_PARAMETER`, and `FIELD`. It should also define an element named `value` (and not `values`), whose type should be a `String` array and have a default value.

25.6 *(d)*

Annotations are not reflected in the documentation of the class in which they are applied, unless the `@Documented` meta-annotation is applied to its annotation type.

25.7 (b) and (e)

Default values are not mandatory for annotation type elements, and a default value cannot be `null`. An annotation element of an array type can be assigned a single default value, in which case it does not need to be enclosed in block notation `{}`. Annotation element names must be specified as method names with no parameters, but mandatory parentheses `()`.

25.8 (a), (b), and (c)

The annotation type can be applied to Java types—that is, classes, enums, and interfaces. It can also be applied to fields and constructors.

(a) applies annotation to an interface, which is valid because it is defined as applicable to a type. It provides a string value for the `value()` element, and it does not set any other elements, relying instead on their default values. This means it does not have to explicitly qualify the element `value()` name.

(b) applies annotation to enum fields, which is allowed by this annotation definition. It provides both `value()` and `details()` element values for the first three fields, and relies on the default value of the `details()` element for the fourth field.

(c) applies the annotation to a field, explicitly qualifying the element `value` and relying on the default value of the `details()` element.

(d) applies annotation to a class, which is allowed by this annotation type. However, it does not qualify the name of the `value()` element, which must be qualified when it is not the only element specified.

(e) applies annotation to a method, which is not allowed by this annotation type.

(f) applies annotation to an expression (i.e., in a *type context*), which is not allowed by this annotation type.

25.9 (b)

The `Containee` annotation type is defined as a repeatable annotation type. Thus it can be used as an array type of the mandatory `value()` element declared in the `Container` annotation type. However, in order for this construct to work, any other elements specified in the `Container` annotation type must be declared with default values.

25.10 (c)

The way in which the annotation `@Folder` is applied to the `Storage` class requires the-`Folder` annotation type to be repeatable—that is, its type declaration must be marked with the `@Repeatable` meta-annotation having the argument `Folders.class`. Because of the way the `@Folder` annotation is applied to the `Storage` class, its type declaration must define two elements: a `value()` element and a `temp()` element. The `temp()` element of the `Folder` annotation type must be defined with a default value. The container annotation type `Folders` must define a `value()` element of type `Folder[]`.

26 Secure Coding

26.1 *(c)*

Data obfuscation is considered to be an important measure for securing sensitive information. Sanitizing input values is a countermeasure that helps preventing code injections. Encapsulation helps to prevent code corruption. Terminating recursive data references helps to prevent a type of denial-of-service attack that attempts to cause a program to start an infinite data processing loop.

26.2 *(d)*

Encapsulation is a software design strategy that only allows access to an object’s state through specifically defined operations. Mutable objects are those whose state can be modified. Code corruption is a category of threats that can be used to corrupt application logic. A code injection allows executable code to be passed as an input parameter.

26.3 *(d)*

Normally an addition of 1 to a maximum value of a primitive type would result in the value wrapping around to the minimum value. The maximum value of the `int` type is `2147483647` and the minimum value is `-2147483648`. However, in this case, the addition is performed by the `addExact()` method, which actually checks value boundaries and will throw an `ArithmeticException` to prevent the value wrapping around.

26.4 *(c)*

The Secure Hash Algorithm (SHA) is designed to produce a fixed-length result, known as a message digest, from a variable-length input. The number 256 is the length of the digest—that is, the result produced by the hash algorithm.

26.5 *(a)*

Java security policies are specified as a grant, defining restrictions and permissions on code execution and on access to resources.

26.6 *(c)*

Any checked exceptions thrown within the `PrivilegedAction.run()` method must be handled within this method. Alternatively, if checked exceptions need to be propagated outside the `run()` method, then the `PrivilegedExceptionAction` interface should be implemented instead.

Appendix E

Mock Exam: Java SE 17 Developer

This is a mock exam for the *Java SE 17 Developer* exam. It comprises brand-new questions, which are similar to the questions that can be expected on the real exam. Working through this exam will give a good indication of how well you are prepared for the real exam, and whether any topics need further study.

Considering the vast number of Java topics in the exam objectives and only 50 multiple-choice questions on the exam, this mock exam contains questions on a selected number of topics, as on the real exam.

Annotated answers to the questions can be found in Appendix F, p. 1737.

Questions

Q1 What will be printed when the following program is run?

Click here to view code image

```
class Base {
  protected int i;
  Base() { add(1); }
  void add(int v) { i += v; }
  void print() { System.out.println(i); }
}

class Extension extends Base {
  Extension() { add(2); }
  void add(int v) { i += v*2; }
}
```

```
public class Qd073 {
  public static void main(String[] args) {
    bogo(new Extension());
  }

  static void bogo(Base b) {
    b.add(8);
    b.print();
  }
}
```

Select the one correct answer.

a. 9
b. 11
c. 13
d. 21
e. 22

Q2 What will be printed when the following program is executed?

Click here to view code image

```
public class Qcb90 {
  private int a;
  private int b;
  public void f() {
    a = 0;
    b = 0;
    int[] c = { 0 };
    g(b, c);
    System.out.println(a + " " + b + " " + c[0] + " ");
  }

  public void g(int b, int[] c) {
    a = 1;
    b = 1;
    c[0] = 1;
  }

  public static void main(String[] args) {
    Qcb90 obj = new Qcb90();
    obj.f();
  }
}
```

Select the one correct answer.

a. 0 0 0
b. 0 0 1
c. 0 1 0
d. 1 0 0
e. 1 0 1

Q3 Given the following interface declaration, which declaration is valid?

```
interface I {
  void setValue(int val);
  int getValue();
}
```

Select the one correct answer.

Click here to view code image

a.
```
class A extends I {
  int value;
  void setValue(int val) { value = val; }
  int getValue() { return value; }
}
```

b.
```
interface B extends I {
  void increment();
}
```

c. Click here to view code image

```
abstract class C implements I {
  int getValue() { return 0; }
  abstract void increment();
}
```

d.
```
interface D implements I {
  void increment();
}
```

e. Click here to view code image

```
class E implements I {
  int value;
  public void setValue(int val) { value = val; }
}
```

Q4 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Q200A80 {
  public static void main(String[] args) {
    callType(10);
  }

  private static void callType(Number num){
    System.out.println("Number passed");
  }

  private static void callType(Object obj){
    System.out.println("Object passed");
  }
}
```

Select the one correct answer.

a. The program will compile and will print `Object passed`.

b. The program will compile and will print `Number passed`.

c. The program will fail to compile, because the call to the `callType()` method is ambiguous.

d. The program will compile, but it will throw a `ClassCastException` at runtime.

Q5 Given the following code:

Click here to view code image

```
class MyClass {
  public static void main(String[] args) {
    int k = 0;
    int l = 0;
    for (int i = 0; i <= 3; i++) {
      k++;
      if (i == 2) break;
      l++;
    }
    System.out.println(k + ", " + l);
  }
}
```

Which of the following statements is true?
Select the one correct answer.

a. The program will fail to compile.

b. The program will print `3, 3` at runtime.

c. The program will print `4, 3` at runtime if the `break` statement is replaced by the `continue` statement.

d. The program will fail to compile if the `break` statement is replaced by the `return` statement.

e. The program will fail to compile if the `break` statement is replaced by an empty statement.

Q6 Given the declaration:

Click here to view code image

```
int[][] nums = {{20}, {30}, {40}};
```

Which code will compile and print `90` at runtime?
Select the one correct answer.

Click here to view code image

a.

```
{
  int sum = 0;
  for (int[] row : nums[])
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

b. Click here to view code image

```
{
  int sum = 0;
  for (int[] row : nums[][])
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

c. Click here to view code image

```
{
  int sum = 0;
  for (int[] row : nums)
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

d. Click here to view code image

```
{
  int sum = 0;
  for (int[] row : nums)
    for (int val : row)
      sum += val;
  System.out.println(sum);
}
```

e. Click here to view code image

```
{
  int sum = 0;
  for (Integer[] row : nums)
    for (int val : row)
      sum += val;
  System.out.println(sum);
}
```

Q7 What will be the result from running the following program?

Click here to view code image

```
public class Q1408b {
  public static void main(String[] args) {
    int i = 0;
    while (++i == i) {
      System.out.println(i++);
    }
  }
}
```

Select the one correct answer.

a. The program will execute and terminate normally, but it will not print anything.

b. The program will execute indefinitely, printing all numbers from 1 onward.

c. The program will execute indefinitely, printing all numbers from 2 onward.

d. The program will execute indefinitely, printing all even numbers from 2 onward.

e. The program will execute indefinitely, printing all odd numbers from 1 onward.

f. The program will execute indefinitely, printing all odd numbers from 3 onward.

Q8 What will be the result from running the following program?

Click here to view code image

```
public class RemainderFun {
    public static void main(String[] args) {
      int i = 24, k = 7;
      System.out.print( i %  k + "|");
      System.out.print( i % -k + "|");
      System.out.print(-i %  k + "|");
      System.out.println(-i % -k);
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw an exception at runtime.

c. `3|-3|-3|3`

d. `3|3|-3|-3`

e. `3|-3|-3|-3`

f. `3|-3|3|-3`

Q9 Which statement is true about the following program?

Click here to view code image

```
  public class Switchy {
    public static void main(String[] args) {
      final String s1 = "January";
      final String yr = " 2022";
      s1.concat(yr);
      switch (s1) {
        default -> System.out.println("Sorry.");
        case "January" + yr, s1 + " 2023" -> System.out.println("OK.");
      }
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile. When run, it will print: `Sorry.` `OK.`

c. The program will compile. When run, it will print: `Sorry.`

d. The program will compile. When run, it will print: `OK.`

Q10 What will be the result from the following program?

Click here to view code image

```
  public class CodeMe {
    public static void main(String[] args) {
      boolean flag = false;
      if (false)          // (1)
        flag = !flag;
      System.out.println(flag);
    }
  }
```

Select the two correct answers.

a. The program, as it stands, does not compile.

b. The program will compile without errors. When run, it will print `false`.

c. The program will compile without errors. When run, it will print `true`.

d. If the keyword `if` at (1) is replaced with the keyword `while`, the program will compile without errors. When run, it will print `false`.

e. If the keyword `if` at (1) is replaced with the keyword `while`, the program will compile without errors. When run, it will print `true`.

f. If the keyword `if` at (1) is replaced with the keyword `while`, the program will fail to compile.

Q11 What will be the result from the following program?

Click here to view code image

```
import java.util.List;

class Person implements Comparable<Person> {
  public int compareTo(Person p) {
    return 1;
  }
}

class Student extends Person {
  public int compareTo(Student s) {
    return 2;
  }
}

public class Calling {
  public static void main(String[] args) {
    Person  p1 = new Person();
    Student s1 = new Student();
    Student s2 = new Student();
    Person  p2 = s1;
    System.out.println(List.of(
        p1.compareTo(s1), p1.compareTo(p2),
        p2.compareTo(s1), p2.compareTo(p1),
        s1.compareTo(p1), s1.compareTo(p2),
        s1.compareTo(s2)));
  }
}
```

Select the one correct answer.

a. `[1, 1, 2, 1, 1, 1, 2]`
b. `[1, 1, 1, 1, 1, 1, 2]`
c. `[1, 1, 2, 1, 1, 2, 2]`
d. `[1, 1, 1, 1, 1, 2, 2]`
e. `[1, 1, 1, 1, 2, 2, 2]`

Q12 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class Q12A56 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add(0, "Ada");
    strList.add("Alyla");
    strList.set(strList.size()-1, "Otto");
    strList.add(strList.size()-1, "Anna");
    System.out.println(strList);                             // (1)
    int size = strList.size();
    for (int i = 0; i < size; ++i) {
      strList.add(strList.get(size-1-i));
    }
    System.out.println(strList);                             // (2)
  }
}
```

Select the two correct answers.

- **a.** (1) will print `[Ada, Alyla, Anna]`.
- **b.** (1) will print `[Ada, Anna, Otto]`.
- **c.** (1) will print `[Ada, Otto, Alyla]`.
- **d.** (2) will print `[Ada, Alyla, Anna, Anna, Alyla, Ada]`.
- **e.** (2) will print `[Ada, Anna, Otto, Otto, Anna, Ada]`.
- **f.** (2) will print `[Ada, Otto, Alyla, Alyla, Otto, Ada]`.

Q13 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;

public class Q12A55 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add(strList.size(), "Anna");
    strList.add(strList.size()-1, "Ada");
    strList.add(strList.size()-1, "Otto");
    strList.add(0, "Alyla");
    System.out.println(strList);
    int size = strList.size();
    for (int i = 0; i < size/2; ++i) {
      String strTemp = strList.get(i);
      strList.set(i, strList.get(size-1-i));
      strList.set(size-1-i, strTemp);
    }
    System.out.println(strList);
  }
}
```

Select the one correct answer.

- **a.** The program will fail to compile.
- **b.** The program will throw an `IndexOutOfBoundsException`.
- **c.** The program will throw a `NullPointerException`.
- **d.** The program will print:

```
[Alyla, Ada, Otto, Anna]
```

```
[Anna, Otto, Ada, Alyla]
```

e. The program will print:

```
[Ada, Otto, Alyla, Anna]
[Anna, Alyla, Otto, Ada]
```

Q14 Given the following code:

Click here to view code image

```
import java.util.function.Predicate;
public class Test {
  public static void main(String[] args) throws IOException  {

    Stream<String> shapes = Stream.of("Circle","Cube");
    // (1) INSERT CODE HERE
    shapes.filter(p).forEach(v->System.out.println(v));
  }
}
```

Which code option can be inserted at (1) so that the program prints `"Circle"` and `"Cube"`?
Select the one correct answer.

a. Click here to view code image

```
Predicate<String> p = s -> {
  s.toLowerCase();
  return s.contains("c");
    };
```

b. Click here to view code image

```
Predicate p = s -> {
  s = s.toString().toLowerCase();
  return s.contains("c");
    };
```

c. `Predicate<String> p = s -> s.contains("c");`

d. `Predicate<Object> p = s -> s.toString().toLowerCase().contains("c");`

e. `Predicate p = s -> s.contains("c");`

f. `Predicate p = s -> s.toLowerCase().contains("c");`

Q15 Which statement is true about the following interfaces?

Click here to view code image

```
interface IA            { boolean equals(Object obj); }
interface IB extends IA { boolean doIt(String str); }
interface IC extends IB { boolean doIt(String str); }
interface ID extends IC { boolean equals(Object obj);}
```

Select the one correct answer.

a. `IA` is a functional interface.

b. `IB` is a functional interface.

c. `IC` is not a functional interface.

d. `ID` is not a functional interface.

Q16 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.time.LocalDate;

public class Q11A35 {
  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2015, 1, 1);
    date.withYear(5);
    System.out.println(date.plusMonths(12));
  }
}
```

Select the one correct answer.

a. The program will fail to compile.
b. The program will throw an exception at runtime.
c. The program will print `0006-01-01`.
d. The program will print `2021-01-01`.
e. The program will print `2015-01-01`.
f. The program will print `2016-01-01`.

Q17 Which code, when inserted at (1), will make the program compile and execute normally?

Click here to view code image

```
import java.time.*;
import java.time.format.*;

public class Q11A95 {
  public static void main(String[] args) {
    String inputStr = "The time is 15 minutes past 10PM.";
    String pattern = "'The time is 'm' minutes past 'ha.";
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern(pattern);
    // (1) INSERT CODE HERE
  }
}
```

Select the three correct answers.

a. `LocalTime time = LocalTime.parse(inputStr, dtf);`
b. `LocalDate date = LocalDate.parse(inputStr, dtf);`
c. `LocalDateTime dateTime = LocalDateTime.parse(inputStr, dtf);`
d. `String timeStr = LocalTime.of(9, 20).format(dtf);`
e. `String dateStr = LocalDate.of(2015, 12, 24).format(dtf);`
f. `String dateTimeStr = LocalDateTime.of(2015, 12, 24, 22, 15).format(dtf);`

Q18 Given that a static method `doIt()` in the class `Work` represents work to be done, which lines of code will succeed in starting a new thread that will do the work?

Select the one correct answer.

a. Click here to view code image

```
Runnable r = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
Thread t = new Thread(r);
   t.start();
```

b. Click here to view code image

```
Thread t = new Thread() {
  public void start() {
    Work.doIt();
  }
};
t.start();
```

c. Click here to view code image

```
Runnable r = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
r.start();
```

d. Click here to view code image

```
Thread t = new Thread(new Work());
t.start();
```

e. Click here to view code image

```
Runnable t = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
t.run();
```

Q19 Which of the following statements are true about the following program?

Click here to view code image

```
public class Q100A82 {
  public static void main(String[] args) {
    Object o = choose(991, "800");                                  // (1
    Number n1 = choose(991, 3.14);                                  // (2
    Number n2 = Q100A82.<Double>choose((double)991, 3.14);          // (3
    int k = (int) choose(1.3, 3.14);                                // (4
    int l = (int) (double) choose(1.3, 3.14);                       // (5
  }

  public static <T extends Comparable<T>> T choose(T t1, T t2) {
    return t1.compareTo(t2) >= 0 ? t1 : t2;
  }
}
```

Select the two correct answers.

- **a.** The class must be declared as a generic type: public class Q100A82<T extends Comparable<T>> { ... }
- **b.** The compiler reports errors at (1).
- **c.** The compiler reports no errors at (2).
- **d.** The compiler reports no errors at (3).
- **e.** The compiler reports no errors at (4).
- **f.** The compiler reports errors at (5).

Q20 Which of the following statements, when inserted at (1), will make the program print `1` when executed?

Click here to view code image

```
public class Outer {
  private int a = 1;
  private int b = 1;
  private int c = 1;

  class Inner {
    private int a = 2;
    int get() {
      int c = 3;
      // (1) INSERT CODE HERE
      return c;
    }
  }
  Outer() {
    Inner i = new Inner();
    System.out.println(i.get());
  }

  public static void main(String[] args) {
    new Outer();
  }
}
```

Select the two correct answers.

- **a.** `c = b;`
- **b.** `c = this.a;`
- **c.** `c = this.b;`
- **d.** `c = Outer.this.a;`
- **e.** `c = c;`

Q21 What will be the result of compiling and running the following code?

Click here to view code image

```
public enum FrequentFlyer {
  PLATINUM(20), GOLD(10), SILVER(5), BASIC(0);
  private double extra;

  FrequentFlyer(double extra) {
    this.extra = extra;
  }

  public static FrequentFlyer max(FrequentFlyer c1, FrequentFlyer c2) {
    return c1.compareTo(c2) < 0 ? c2 : c1;
  }

  public static FrequentFlyer max2(FrequentFlyer c1, FrequentFlyer c2) {
    return c1.extra < c2.extra ? c2 : c1;
  }

  public static void main (String[] args) {
    System.out.println(GOLD.ordinal() > SILVER.ordinal());
    System.out.println(max(GOLD, SILVER));
    System.out.println(max2(GOLD, SILVER));
  }
}
```

Select the one correct answer.

a. The program will compile and print:

```
false
SILVER
GOLD
```

b. The program will compile and print:

```
true
GOLD
SILVER
```

c. The program will compile and print:

```
true
GOLD
GOLD
```

d. The program will fail to compile, since the enum type `FrequentFlyer` does not implement the `Comparable` interface.

Q22 Which of the following statements are true about the following code?

Click here to view code image

```
public class Vertical {
  private int alt;
  public synchronized void up() {
    ++alt;
  }
  public void down() {
    --alt;
  }
  public synchronized void jump() {
    int a = alt;
    up();
    down();
    System.out.println(a == alt);
  }
}
```

Select the two correct answers.

a. The code will fail to compile.

b. Different threads can execute the `up()` method concurrently.

c. Different threads can execute the `down()` method concurrently.

d. Different threads can execute both the `up()` and the `down()` methods concurrently.

e. The `jump()` method will always print `true`.

Q23 Which parameter declarations can be inserted at (1) so that the program will compile without warnings?

Click here to view code image

```
interface Wagger{}
class Pet implements Wagger{}
class Dog extends Pet {}
class Cat extends Pet {}
public class Q100A51 {
  public static void main(String[] args) {
    List<Pet> p = new ArrayList<>();
    List<Dog> d = new ArrayList<>();
    List<Cat> c = new ArrayList<>();
    examine(p);
    examine(d);
    examine(c);
  }

  static void examine(/* INSERT PARAMETER TYPE HERE */ pets) {       // (
    System.out.print("Your pets need urgent attention.");
  }
}
```

Select the three correct answers.

a. `List<? extends Pet>`

b. `List<? super Pet>`

c. `List<? extends Wagger>`

d. `List<? super Wagger>`

e. `List<?>`

Q24 Given the following code:

Click here to view code image

```
package p1;
public enum Format {
  JPEG, GIF, TIFF;
}
```

```
package p1;
public class Util {
  public enum Format {
    JPEG { public String toString() {return "Jpeggy"; }},
    GIF  { public String toString() {return "Giffy"; }},
    TIFF { public String toString() {return "Tiffy"; }};
  }
  public static <T> void print(T t) {
    System.out.print("|" + t + "|");
  }
}
```

```
import static p1.Util.Format;
import static p1.Util.print;

public class Importing {
  static final int JPEG = 200;
  public static void main(String[] args) {
    final int JPEG = 100;
    print(JPEG);
    // (1) INSERT CODE HERE
  }
}
```

Which lines of code when inserted at (1) will result in the following output:

```
|100||200||Jpeggy||JPEG|
```

Select the one correct answer.

a. `print(Format.JPEG);`
`print(p1.Format.JPEG);`
`print(Importing.JPEG);`

b. `print(Format.JPEG);`
`print(Importing.JPEG);`
`print(p1.Format.JPEG);`

c. `print(Importing.JPEG);`
`print(Format.JPEG);`
`print(p1.Format.JPEG);`

Q25 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.ArrayList;
import java.util.List;
public class Q400A70 {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(2019); list.add(2020); list.add(2021);
    System.out.println("Before: " + list);
    for (int i : list) {
      int index = list.indexOf(i);
      list.set(index, ++i);
    }
    System.out.println("After:  " + list);
  }
}
```

Select the one correct answer.

a. The program will print:

```
Before: [2019, 2020, 2021]
After: [2020, 2021, 2022]
```

b. The program will print:

```
Before: [2019, 2020, 2021]
After: [2022, 2020, 2021]
```

c. The program will print:

```
Before: [2019, 2020, 2021]
After: [2019, 2020, 2021]
```

d. The program will print:

```
Before: [2019, 2020, 2021]
After: [2020, 2021, 2019]
```

e. The program will throw a `java.util.ConcurrentModificationException` at runtime.

Q26 Given the following code:

Click here to view code image

```
public class Person {
  protected transient String name;
  Person() { this.name = "NoName"; }
  Person(String name) { this.name = name; }
}
```

and

Click here to view code image

```
public class Student extends Person {
  protected long studNum;
  Student() { }
  Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
}
```

and

Click here to view code image

```
import java.io.Serializable;
public class GraduateStudent extends Student implements Serializable {
  private int year;
  GraduateStudent(String name, long studNum, int year) {
    super(name, studNum);
    this.year = year;
  }

  public String toString() {
    return "(" + name + ", " + studNum + ", " + year + ")";
  }
}
```

and

Click here to view code image

```
import java.io.*;
public class Q800A60 {
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat
         ObjectOutputStream outputStream = new ObjectOutputStream(output
      GraduateStudent stud1 = new GraduateStudent("Aesop", 100, 1);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile
      GraduateStudent stud2 = (GraduateStudent) inputStream.readObject()
      System.out.println(stud2);
    }
  }
}
```

Which statement is true about the program?
Select the one correct answer.

- **a.** It will fail to compile.
- **b.** It will compile, but it will throw an exception at runtime.
- **c.** It will print `(Aesop, 100, 1)(NoName, 0, 1)`.
- **d.** It will print `(Aesop, 100, 1)(Aesop, 100, 1)`.
- **e.** It will print `(Aesop, 100, 1)(null, 0, 1)`.

Q27 Which of the following statements are true about the classes `SupA`, `SubB`, and `SubC`?

Click here to view code image

```
public class SupA<T> {
  public List<?> fuddle() { return null; }
  public List scuddle(T t) { return null; }
}
public class SubB<U> extends SupA<U> {
  public List fuddle() { return null;}
  public List<?> scuddle(U t) { return null; }
}
public class SubC<V> extends SupA<V> {
  public List<V> fuddle() { return null;}
  public List<? extends Object> scuddle(V t) { return null; }
}
```

Select the four correct answers.

- **a.** Class `SubB` will fail to compile.
- **b.** Class `SubC` will fail to compile.
- **c.** Class `SubB` will compile.
- **d.** Class `SubC` will compile.
- **e.** Class `SubB` overloads the methods in class `SupA`.
- **f.** Class `SubC` overloads the methods in class `SupA`.
- **g.** Class `SubB` overrides the methods in class `SupA`.
- **h.** Class `SubC` overrides the methods in class `SupA`.

Q28 Which of the following statements is true?
Select the one correct answer.

- **a.** If a method does not handle an exception that is thrown, it must declare the exception in a `throws` clause.
- **b.** A `try` block cannot be followed by both a `catch` and a `finally` clause.
- **c.** An empty `catch` clause is not allowed.
- **d.** A `catch` clause cannot follow a `finally` clause.
- **e.** A `finally` clause must always follow one or more `catch` clauses.

Q29 Which import statements, when inserted at (4) in package `p3`, will result in a program that can be compiled and run?

Click here to view code image

```
package p2;
enum March {LEFT, RIGHT;                                  // (1)
  public String toString() {
    return "Top-level enum";
  }
}
public class DefenceInDepth {
  public enum March {LEFT, RIGHT;                         // (2)
    public String toString() {
      return "Static enum";
    }
  }
  public enum Military { INFANTRY, AIRFORCE;
    public static enum March {LEFT, RIGHT;                // (3)
      public String toString() {
        return "Statically nested enum";
      }
    }
  }
}
```

```
package p3;
// (4) INSERT IMPORTS HERE
public class MarchingOrders {
  public static void main(String[] args) {
    System.out.println(March.LEFT);
    System.out.println(DefenceInDepth.March.LEFT);
    System.out.println(p2.DefenceInDepth.March.LEFT);
    System.out.println(Military.March.LEFT);
    System.out.println(DefenceInDepth.Military.March.LEFT);
    System.out.println(p2.DefenceInDepth.Military.March.LEFT);
    System.out.println(LEFT);
  }
}
```

Select the three correct answers.

a. Click here to view code image

```
import p2.*;
import p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
```

b. Click here to view code image

```
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
```

c. Click here to view code image

```
import p2.DefenceInDepth;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
```

d. Click here to view code image

```
import static p2.DefenceInDepth;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
```

e. Click here to view code image

```
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.*;
```

f. Click here to view code image

```
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March;
```

Q30 Which statement is true about the following code?

Click here to view code image

```
public class A {
  public A() {}
  public A(int i) { this(); }
}

public class B extends A {
  public boolean B(String msg) { return false; }
}

public class C extends B {
  private C() { super(); }
  public C(String msg) { this(); }
  public C(int i) {}
}
```

Select the one correct answer.

a. The code will fail to compile.

b. The constructor in `A` that takes an `int` as an argument will never be called as a result of constructing an object of class `B` or `C`.

c. Objects of class `B` cannot be constructed.

d. At most, one constructor of each class is called as a result of constructing an object of class `C`.

Q31 Given the following class declarations, which expression identifies whether the object referenced by `obj` was created by instantiating class `B` rather than classes `A`, `C`, and `D`?

```
class A {}
class B extends A {}
class C extends B {}
class D extends A {}
```

Select the one correct answer.

a. `obj instanceof B`

b. `obj instanceof A && !(obj instanceof C)`

c. `obj instanceof B && !(obj instanceof C)`

d. `!(obj instanceof C || obj instanceof D)`

e. `!(obj instanceof A) && !(obj instanceof C) && !(obj instanceof D)`

Q32 What will be the result of compiling and running the following code?

Click here to view code image

```
public enum Scale5 {
  GOOD, BETTER, BEST;

  public char getGrade() {
    return switch (this) {
      case GOOD   -> 'C';
      case BETTER -> 'B';
      case BEST   -> 'A';
    };
  }

  public static void main (String[] args) {
    System.out.println(GOOD.getGrade());
  }
}
```

Select the one correct answer.

- **a.** The program will fail to compile, as the `switch` expression is not compatible with the `case` labels.
- **b.** The program will fail to compile, as enum constants cannot be used as `case` labels.
- **c.** The program will fail to compile, as the `case` labels must be qualified with the enum type name.
- **d.** The program will compile, and when run, will print `C`.
- **e.** The program will compile, and when run, will print `GOOD`.

Q33 Which method declarations, when inserted at (7), will not result in a compile-time error?

Click here to view code image

```
// File: MyClass.java
package p4;
class MySuperclass {
  public        Integer step1(int i)                    { return 1; }
  protected     String  step2(String str1, String str2) { return str1; }
  public        String  step2(String str1)              { return str1; }
  public static String  step2()                         { return "Hi"; }

  public MyClass      makeIt()  { return new MyClass(); }
  public MySuperclass makeIt2() { return new MyClass(); }
}

public class MyClass extends MySuperclass {
  // (7) INSERT METHOD DECLARATION HERE
}
```

Select the two correct answers.

- **a.** `public int step1(int i) { return 1; }`
- **b.** `public String step2(String str2, String str1) { return str1; }`
- **c.** `private void step2() { }`
- **d.** `private static void step2() { }`

e. `private static String step2(String str) { return str; }`

f. `public MySuperclass makeIt() { return new MySuperclass(); }`

g. `public MyClass makeIt2() { return new MyClass(); }`

Q34 What is the result of compiling and running the following program?

Click here to view code image

```
public class Q800A60 {
  static void printFirst(Integer... ints) {
    System.out.println("Integer...: " + ints[0]);
  }

  static void printFirst(Number... nums) {
    System.out.println("Number...: " + nums[0]);
  }

  static void printFirst(Object... objs) {
    System.out.println("Object...: " + objs[0]);
  }

  public static void main(String[] args) {
    printFirst(10);
    printFirst((byte)20);
    printFirst('3', '0');
    printFirst("40");
    printFirst((short)50, 55);
    printFirst((Number[])new Integer[] {70, 75});
  }
}
```

Select the one correct answer.

a. The program does not compile because of ambiguous method calls.

b. The program will compile and will print:

```
Integer...: 10
Integer...: 20
Integer...: 3
Object...: 40
Integer...: 50
Number...: 70
```

c. The program will compile and will print:

```
Integer...: 10
Number...: 20
Object...: 3
Object...: 40
Number...: 50
Number...: 70
```

d. The program will compile and will print:

```
Integer...: 10
Integer...: 20
Integer...: 3
Object...: 40
Number...: 50
Number...: 70
```

Q35 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Initialization {
  private static String msg(String msg) {
    System.out.println(msg);
    return msg;
  }
  static String m = msg("1");
  { m = msg("2"); }
  static { m = msg("3"); }
  public static void main(String[] args) {
    Object obj = new Initialization();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile and print `1`, `2`, and `3` at runtime.

c. The program will compile and print `2`, `3`, and `1` at runtime.

d. The program will compile and print `3`, `1`, and `2` at runtime.

e. The program will compile and print `1`, `3`, and `2` at runtime.

Q36 Given the following resource bundle:

Click here to view code image

```
# File: MyResources_en_US.properties
greeting = Long time no see!
```

Assuming that the current default locale is `"en_US"`, what will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.*;
public class LocatingBundlesForDefaultLocale {
  public static void main(String[] args) {
    Locale norLocale = new Locale("no", "NO");
    ResourceBundle rbs = ResourceBundle.getBundle("MyResources", norLoca
    System.out.println(rbs.getString("greeting"));
    Locale.setDefault(norLocale);
    System.out.println(rbs.getString("greeting"));
  }
}
```

Select the one correct answer.

a. When run, the program will print the following and terminate normally. `Long time no see! Long time no see!`

b. When run, the program will print `Long time no see!` and terminate normally.

c. When run, the program will print `Long time no see!` and then throw a `Missing-ResourceException`.

d. When run, the program will immediately throw a `MissingResourceException`.

e. When run, the program will terminate normally without printing anything.

Q37 What will be the result of compiling and running the following program?

Click here to view code image

```
public class Syncher2 {
  static final int[] intArray = new int[2];

  private static void pause() {
    while (intArray[0] == 0) {
      try { intArray.wait(); }
      catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    }
  }

  public static void main (String[] args) {
    Thread runner = new Thread() {
      public void run() {
        synchronized (intArray) {
          pause();
          System.out.println(intArray[0] + intArray[1]);
    }}};

    runner.start();
    intArray[0] = intArray[1] = 10;
    synchronized(intArray) {
      intArray.notify();
    }
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile. When run, it will throw an exception.

c. The program will compile and continue running once started, but will not print anything.

d. The program will compile. When run, it will print `0` and terminate normally.

e. The program will compile. When run, it will print `20` and terminate normally.

f. The program will compile. When run, it will print some number other than `0` or `20`, and terminate normally.

Q38 Given the following code:

Click here to view code image

```
public class Thingy<T> implements Comparable<T> {
    private T value;
    public Thingy(T value) {
      this.value = value;
    }
    public String toString() {
      return value.toString();
    }
    public int compareTo(T obj) {
      return this.value.toString().compareTo(obj.toString());
    }
  }
```

and

Click here to view code image

```
  import java.math.*;
  import java.util.*;
  public class TestThingy {
    public static void main(String[] args) {
      Thingy[] values  = {
        new Thingy<BigDecimal>(BigDecimal.valueOf(12.99)),
        new Thingy<BigDecimal>(BigDecimal.valueOf(7.99)),
        new Thingy<BigDecimal>(BigDecimal.valueOf(7)),
        new Thingy<BigDecimal>(BigDecimal.valueOf(9.99))};
      Arrays.sort(values);
      for (Thingy t: values) {
        System.out.print(t + " ");
      }
    }
  }
```

What is the result?

Select the one correct answer.

a. `12.99 7.99 7 9.99`

b. `12.99 7 7.99 9.99`

c. `7 7.99 9.99 12.99`

d. `12.99 9.99 7.99 7`

e. The code will throw an exception at runtime.

f. The code will fail to compile.

Q39 Given the following code:

Click here to view code image

```
  public class Widget<T> {
    private T value;
    private static int result;
    public Widget(T value) { this.value = value; }
    public void compute(Comparable<T> c) {
      result += c.compareTo(value);
    }
    public static int getResult() { return result; }
  }
```

and

Click here to view code image

```
  import java.time.*;
  public class TestWidget {
    public static void main(String[] args) {
      Widget<String> t1 = new Widget<>("ACME");
      t1.compute(v -> v.length());
    t1.compute(v -> v.indexOf("C"));
    Widget<LocalDate> t2 = new Widget<>(LocalDate.of(2020, 10, 20));
    t2.compute(v -> v.getDayOfMonth());
    System.out.println(Widget.getResult());
  }
}
```

What is the result?
Select the one correct answer.

a. 23
b. 24
c. 25
d. 13
e. 14
f. 15
g. The code will throw an exception at runtime.
h. The code will fail to compile.

Q40 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test13A10 {
  public static void main(String[] args) {
    Integer[] values = {4, 2, 6, 3, 5};
    Arrays.sort(values, (x, y) -> x - y);
    System.out.println(Arrays.toString(values));
    }
  }
}
```

What is the result?
Select the one correct answer.

a. [4, 2, 6, 3, 5]
b. [2, 3, 4, 5, 6]
c. [6, 5, 4, 3, 2]
d. [2, -4, 3, -2]
e. The code will throw an exception at runtime.
f. The code will fail to compile.

Q41 Given the following code:

Click here to view code image

```
import java.time.*;
import java.time.format.DateTimeFormatter;
public class Q700A12 {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(0, 1, 1);
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd MM yy G");
    System.out.println(d.format(dtf));
  }
}
```

What is the result?
Select the one correct answer.

a. `01 01 01 BC`
b. `00 01 01 BC`
c. `01 01 01 AD`
d. `00 01 01 AD`

Q42 Given the following code:

Click here to view code image

```
Comparable<?>[] values = { LocalDate.of(2022, Month.JANUARY, 1),
    LocalDate.of(2022, Month.FEBRUARY, 5),
    Integer.valueOf(3),
    LocalDateTime.of(2022, Month.JANUARY, 22, 1, 12, 2),
    "4"};
long result = Arrays.stream(values)
    .mapToInt(v -> (v instanceof LocalDateTime ldt)
                    ? ldt.toLocalTime().getSecond() : 1)
    .sum();
System.out.println(result);
```

What is the result?
Select the one correct answer.

a. `2`
b. `4`
c. `6`
d. `8`
e. The code will throw an exception at runtime.

Q43 Given the following code:

Click here to view code image

```
String txt = """
  a
    b
  c
  """;
txt.lines().map(String::length).forEachOrdered(System.out::print);
```

The first line of the text block has two leading spaces, the second line has four leading spaces, and the third line has two leading spaces. There are two leading whitespace on the line with the closing delimiter of the text block.
What is the result of executing the code?
Select the one correct answer.

a. `353`
b. `3530`
c. `1510`
d. `131`

Q44 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class Test {
  public static void main(String[] args) {
    Path p1 = Path.of("/users/joe");
    Path p2 = Path.of("/users/bob");
    Path p3 = p1.resolve(p1.relativize(p2));
    Path p4 = p3.normalize();
    System.out.println(p3.getName(1) + " " + p4.getName(1));
  }
}
```

What is the result?
Select the one correct answer.

a. `users bob`
b. `joe users`
c. `joe bob`
d. `users users`

Q45 Given the following code:

Click here to view code image

```
import java.util.*;
public class Localizing {  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("resources", new Locale
    DateTimeFormatter fmt = DateTimeFormatter.ofPattern(rb.getString("f1
                                                        new Locale("en",
    LocalDate d = LocalDate.of(2022, 4, 1);
    System.out.println(fmt.format(d));
  }
}
```

and

Click here to view code image

```
// File: resources.properties
f1=yy-MMM-dd
```

and

Click here to view code image

```
// File: resources_en.properties
f2=MMM-dd-yy
```

and

Click here to view code image

```
// File: resources_en_GB.properties
f1=dd-MMM-yy
```

What is the result?
Select the one correct answer.

a. `01-Apr-22`

b. `22-Apr-01`

c. `Apr-01-22`

d. The code will throw an exception at runtime.

Q46 Given the following code:

Click here to view code image

```
public class TestX {
  public static void main(String[] args) {
    System.out.println(action(Integer.valueOf("1")));
  }
  public static boolean action(Object obj) {
    return (obj instanceof String value1 && value1.contains("1") ||
            obj instanceof Integer value2 && value2.intValue() < 1);
    }
}
```

What is the result?
Select the one correct answer.

a. `true`

b. `false`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

Q47 Given the following code:

Click here to view code image

```
import java.time.Duration;
public record Song(String title, Duration duration) {
  // (1) INSERT CODE HERE
}
```

and

Click here to view code image

```
public class TestY {
  public static void main(String[] args) {
    Song song = new Song("Imagine", 106);
    System.out.println(song);
  }
}
```

Which option can be inserted into the `Song` record declaration at (1), so that the program will print `Song[title=IMAGINE, duration=PT1M46S]`?
Select the one correct answer.

a. Click here to view code image

```
 public Song(int seconds) {
      this(this.title.toUpperCase(), Duration.ofSeconds(seconds));
   }
```

b. Click here to view code image

```
public Song(int seconds) {
     this(title.toUpperCase(), Duration.ofSeconds(seconds));
   }
```

c. Click here to view code image

```
public Song(String title, int seconds) {
     this(title.toUpperCase(), Duration.ofSeconds(seconds));
   }
```

d. Click here to view code image

```
public Song(String title, int seconds) {
     this.title = title.toUpperCase();
     this.duration = Duration.ofSeconds(seconds);
   }
```

e. Click here to view code image

```
public Song {
     this.title.toUpperCase();
     this.duration = Duration.ofSeconds(seconds));
   }
```

Q48 Given the following directory structure under the directory `./Sun`:

Click here to view code image

```
./Sun/1_Mercury
./Sun/2_Venus
./Sun/3_Earth
./Sun/3_Earth/1_Moon
./Sun/4_Mars
./Sun/4_Mars/1_Phobos
./Sun/4_Mars/2_Deimos
./Sun/5_Jupiter
./Sun/5_Jupiter/1_Io
./Sun/5_Jupiter/2_Europa
./Sun/5_Jupiter/3_Ganymede
./Sun/5_Jupiter/4_Calisto
./Sun/6_Saturn
./Sun/7_Uranus
./Sun/8_Neptune
```

and the code:

Click here to view code image

```
try {
  Path sun = Path.of("./Sun");
  Files.walk(sun)
       .map(p->p.getName(p.getNameCount()-1).toString())
       .sorted()
       .limit(3)
       .forEach(p->System.out.println(p.substring(2)));
} catch (IOException e) {
  e.printStackTrace();
}
```

What is the result?

Select the one correct answer.

a.

```
Io
Mercury
```

b.

```
Io
Mercury
Moon
```

c.

```
Io
Mercury
Moon
Phobos
```

d.

```
Earth
Ganymede
```

e.

```
Venus
Deimos Europa
```

Q49 Given the text file `data.txt` that contains following lines:

```
Apple:1.99
Pear:1.70
Apple:1.70
Apple:1.75
Orange:1.99
```

and the following code:

Click here to view code image

```
try {
  Path test = Path.of("data.txt");
  Map<String, List<String>> values
      = Files.lines(test).collect(Collectors.groupingBy(
                                           ln -> ln.substring(ln.indexOf(":")
  System.out.println(values);
} catch (IOException e) {
  e.printStackTrace();
}
```

What is the result?
Select the one correct answer.

a. `{1.70=[1.70, 1.70], 1.99=[1.99, 1.99], 1.75=[1.75]}`

b. `{1.70=[Pear:1.70, Apple:1.70], 1.99=[Apple:1.99, Orange:1.99], 1.75=[Apple:1.75]}`

c. `{1.70=[Pear, Apple], 1.99=[Apple, Orange], 1.75=[Apple]}`

d. `{Apple=[Apple:1.99, Apple:1.70, Apple:1.75], Pear=[Pear:1.70], Orange=[Orange:1.99]}`

e. `{Apple=[1.99, 1.70, 1.75], Pear=[1.70], Orange=[1.99]}`

f. `{Apple=[Apple, Apple, Apple], Pear=[Pear], Orange=[Orange]}`

Q50 Given the following code:

Click here to view code image

```
module travel {
  requires transport;
  uses transport.mode.TransportType;
}
```

and

```
module transport {
  exports transport.mode;
}
```

What is the correct declaration for a module named `basic` that implements the `TransportType` service?
Select the one correct answer.

a. Click here to view code image

```
module basic {
  requires travel;
  provides travel.mode.TransportType with basic.mode.Horse;
}
```

b. Click here to view code image

```
module basic {
  requires transport.mode;
  provides TransportType with Horse;
}
```

c. Click here to view code image

```
module basic {
  requires transport;
  provides transport.mode.TransportType with basic.mode.Horse;
}
```

d. Click here to view code image

```
module basic {
  requires travel.mode;
  provides TransportType with Horse;
}
```

Appendix F

Annotated Answers to Mock Exam

This appendix provides annotated answers to the questions in the mock exam for the *Java SE 17 Developer* exam found in Appendix E, p. 1709.

Annotated Answers

Q1 *(e)*

An object of the class `Extension` is created. The first thing the constructor of `Extension` does is invoke the constructor of `Base`, using an implicit `super()` call. All calls to the method `void add(int)` are dynamically bound to the `add()` method in the `Extension` class, since the actual object is of type `Extension`. Therefore, this method is called by the constructor of `Base`, the constructor of `Extension`, and the `bogo()` method with the parameters `1`, `2`, and `8`, respectively. The instance field `i` changes value accordingly: `2`, `6`, and `22`. The final value of `22` is printed.

Q2 *(e)*

Method `g()` modifies field `a`. Method `g()` modifies parameter `b`, not field `b`, since the parameter declaration shadows the field. Variables are passed by value, so the change of value in parameter `b` is confined to the method `g()`. Method `g()` modifies the array whose reference value is passed as a parameter. A change to the first element is visible after return from the method `g()`.

Q3 *(b)*

Classes cannot extend interfaces; they can implement them. Interfaces can extend, but not implement other interfaces. A class must be declared as `abstract` if it does not provide an implementation for all abstract methods of the interfaces that it implements. Instance methods that have no implementation in an interface are implicitly `public` and `abstract`. Classes that implement these methods must explicitly declare these methods to be `public`.

Q4 *(b)*

The method with the most specific signature is chosen. In this case the `int` argument `10` is boxed to an `Integer`, which is passed to the `Number` formal parameter, as type `Number` is more specific than `Object`.

Q5 *(c)*

As it stands, the program will compile correctly and will print `3, 2` at runtime. If the `break` statement is replaced with a `continue` statement, the loop will perform all four iterations and will print `4, 3`. If the `break` statement is replaced with a `return` statement, the whole method will end when `i` equals 2, before anything is printed. If the `break` statement is simply removed, leaving the empty statement (`;`), the loop will complete all four iterations and will print `4, 4`.

Q6 *(d)*

The type of `nums` is `int[][]`. The outer loop iterates over the rows, so the type of the loop variable in the outer loop must be `int[]`, and the loop expression is `nums`. The inner loop iterates over each row, `int[]`. The loop variable in the inner loop must be `int`, and the loop expression in the inner loop is a row given by the loop variable of the outer loop. Only in the loop headers in (d) are both element types compatible.

Q7 *(e)*

The loop condition `++i == i` is always true, as we are comparing the value of `i` to itself, and the loop will execute indefinitely. The evaluation of the loop condition proceeds as follows: (`(++i) == i`), with the operands having the same value. For each iteration, the loop variable `i` is incremented twice: once in the loop condition and a second time in the parameter expression `i++`. However, the value of `i` is printed before it is incremented the second time, resulting in odd numbers from 1 onward being printed. If the prefix operator is also used in the `println` statement, all even numbers from 2 onward would be printed.

Q8 *(d)*

The expression `i % k` evaluates to the remainder value `3`. The expression `i % -k` also evaluates to the remainder value `3`. We ignore the sign of the operands, and negate the remainder only if the dividend (`i` in this case) is negative.

Q9 *(c)*

Strings are immutable, so the method `concat()` does not change the state of the `s1` string. The `default` case is executed in the `switch` statement. There is no fall-through in the `switch` statement, as the switch statement uses arrow notation in which the case labels are mutually exclusive.

Q10 *(b) and (f)*

In both cases, the code in the `if` statement and the `while` loop is unreachable, so it can never be executed. In the case of the `while` loop, the compiler flags an error. The `if` statement is treated as a special case by the compiler to simulate conditional compilation, allowing code that should not be executed.

Q11 *(b)*

The thing to note is that the method `compareTo()` is overloaded in the subclass `Student`, and is not overridden. Thus objects of class `Student` have two methods with the same name: `compareTo`. For overloaded methods, the method to be executed is determined at compile time, depending on the type of the reference used to invoke the method, and the type of the actual parameters. When the type of the reference is `Person` (as is the case for `p1` and `p2`), the method `compareTo()` in `Person` will always be executed. When the type of the reference is `Student` and the argument type is `Person`, the overridden method `compareTo()` in `Person` will always be executed. The overloaded method `compareTo()` defined in the subclass `Student` is executed by the last call `s1.compareTo(s2)` in the `main()` method, where the type of the reference is `Student` and the argument type is also `Student`.

Q12 *(b) and (e)*

The `add(element)` method adds an element at the end of the list. The `add(index, element)` method adds the element at the specified index in the list, shifting elements to the right from the specified index. The index satisfies `(index >= 0 && index <= size())`. The `set(index, element)` method replaces the element at the specified index in the list with the specified element. The index satisfies `(index >= 0 && index < size())`. The `for(;;)` loop adds the elements currently in the list at the end of the list. The list changes as follows:

Click here to view code image

```
[Ada]
[Ada, Alyla]
[Ada, Otto]
[Ada, Anna, Otto]
[Ada, Anna, Otto, Otto, Anna, Ada]
```

Q13 (d)

The `add(index, element)` method accepts an index that satisfies the condition `(index >= 0 && index <= size())`. The `for(;;)` loop swaps elements to reverse the elements in the list.

Q14 (d)

In (a), the type of parameter `s` is inferred to be `String`. The new string returned by calling the `toLowerCase()` method is discarded. Only the string element `"Circle"` contains the substring `"c"` and is printed.

In (b), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The expression `s.contains("c")` in the `return` statement fails to compile since the method `contains()` is not defined in the class `Object`.

In (c), the type of parameter `s` is inferred to be `String`. Only the string element `"Circle"` contains the substring `"c"` and is printed.

In (d), the type of parameter `s` is inferred to be `Object`. The string elements are referenced by the parameter `s` of type `Object`. However, the chain of method calls is invoked on objects of type `String`, resulting in both stream elements being selected and printed.

In (e), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The statement fails to compile since the method `contains()` is not defined in the class `Object`.

In (f), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The class `Object` does not define the method `toLowerCase()` and the statement fails to compile.

Q15 (b)

A functional interface is an interface that has only one abstract method, aside from the abstract method declarations of `public` methods from the `Object` class. This single abstract method declaration can be the result of inheriting multiple declarations of the abstract method from superinterfaces.

All except `IA` are functional interfaces. `IA` does not define an abstract method, as it provides only an abstract method declaration of the concrete `public` method `equals()` from the `Object` class. `IB` defines a single abstract method, `doIt()`. `IC` overrides the abstract method from `IB`, so effectively it has only one abstract method. `IC` inherits the abstract method `doIt()` from `IB` and overrides the `equals()` method from `IA`, so effectively it also has only one abstract method.

Q16 (f)

The date value `2015-01-01` in the `date` reference never changes. The `withYear()` method returns a new `LocalDate` object (with the date value `0005-01-01`) that is ignored. The `plusMonths()` method also returns a new `LocalDate` object whose value is printed. The calculation of `date.plusMonths(12)` proceeds as follows: `2021-01-01 + 12 months (i.e., 1 year) ==> 2022-01-01`

Q17 (a), (d), and (f)

The input string matches the pattern. The input string specifies the time-based values that can be used to construct a `LocalTime` object in (a) by a formatter, based on the time-related pattern letters in the pattern. No date-based values can be interpreted from the input string, as this pattern has only time-related pattern letters. (b) and (c), which require a date part, will throw a `DateTimeParseException`.

To use the pattern for formatting, the temporal object must provide values for the parts corresponding to the pattern letters in the pattern. The `LocalTime` object in (d) has the time part required by the pattern. The `LocalDate` object in (e) does not have the time part required by the pattern, so an `UnsupportedTemporalTypeException` will be thrown. The `LocalDateTime` object in (f) has the time part required by the pattern. In (f), only the time part of the `LocalDateTime` object is formatted.

Q18 (a)

A `Thread` object executes the `run()` method of a `Runnable` object in a separate thread. A `Runnable` object can be provided when constructing a `Thread` object. If no `Runnable` object is supplied, the `run()` method of the `Thread` object (which implements the `Runnable` interface) is executed. A thread is initiated by calling the `start()` method of the `Thread` object.

Q19 (b) and (d)

In (a), the class need not be declared as a generic type if it defines any generic methods.

In (b), the method `choose(T, T)`, where `T` extends `Comparable<T>`, is not applicable to the arguments `(Integer, String)`. Note that `Object` is not `Comparable<Object>`.

In (c), the method `choose(T, T)`, where `T` extends `Comparable<T>`, is not applicable to the arguments `(Integer, Double)`. Note that `Number` is not `Comparable<Number>`.

In (d), the actual type parameter `Double` specified in the method call also requires that the `int` argument is cast to a `double` in order for the call to be valid. The method `choose(T, T)`, where `T` extends `Comparable<T>`, is then applicable to the argument list `(Double, Double)`.

(e) cannot convert the `Double` returned by the method to an `int` using a cast.

In (f), the method returns a `Double` that is first converted to a `double`, which in turn is converted to an `int`.

Q20 (a) and (d)

Field `b` of the outer class is not shadowed by any local or inner class variables; therefore, (a) will work. Using `this.a` will access field `a` in the inner class. Using `this.b` will result in a compile-time error, since there is no field `b` in the inner class. Using `Outer.this.a` will successfully access the field of the outer class. The statement `c = c` will only reassign the current value of the local variable `c` to itself.

Q21 (a)

All enum types implement the `Comparable` interface. Comparison is based on natural order, which in this case is the order in which the constants are specified, with the first one being the smallest. The ordinal value of the first enum constant is 0, the next one has the ordinal value 1, and so on.

Q22 (c) and (d)

Executing synchronized code does not guard against executing non-synchronized code concurrently.

Q23 (a), (c), and (e)

Lists of type `Pet`, `Dog`, and `Cat` are subtypes of `List<? extends Pet>`, `List<? extends Wagger>` and `List<?>`.

`List<? super Pet>` is a supertype for a list of `Pet` itself or a supertype of `Pet`—for example, `Wagger`, but not `Dog` or `Cat`.

`List<? super Wagger>` is a supertype for a list of `Wagger` itself or a supertype of `Wagger`—for example, `Object`, but not `Pet`, `Dog`, or `Cat`.

Q24 (c)

We need to access the following:

- `Importing.JPEG` (to print `200`).
- `p1.Util.Format.JPEG` (to print `Jpeggy`). Since `p1.Util.Format` is statically imported by the second import statement, we need only specify `Format.JPEG`.
- `p1.Format.JPEG` (to print `JPEG`), which is explicitly specified to distinguish it from other `JPEG` declarations.

Q25 (b)

First, note that the `indexOf()` method returns the index of the *first* occurrence of its argument in the list. Although the value of variable `i` is successively changing during the execution of the loop, it is the *first* occurrence of this value that is replaced in each iteration:

Click here to view code image

```
                          0      1     2

                        [2019, 2020, 2021]
  After iteration 1: [2020, 2020, 2021]
  After iteration 2: [2021, 2020, 2021]
  After iteration 3: [2022, 2020, 2021]
```

Note also that we are not removing or adding elements to the list, only changing the reference values stored in the elements of the list.

Q26 (c)

Note that only `GraduateStudent` is `Serializable`. The field `name` in the `Person` class is `transient`. During serialization of a

`GraduateStudent` object, the fields `year` and `studNum` are included as part of the serialization process, but not the field `name`. During deserialization, the default constructors of the superclasses up the inheritance hierarchy of the `GraduateStudent` class are called, as none of the superclasses are `Serializable`.

Q27 (c), (d), (g), and (h)

The method header signature of the corresponding methods is the same after erasure—that is, `List fuddle()` and `List scuddle(Object)`. The return type of overriding methods can be a raw type or a parameterized type.

Q28 (d)

A `try` block must be followed by at least one `catch` or `finally` clause. No `catch` clause can follow a `finally` clause. Methods must declare any checked exceptions in a `throws` clause, if they do not catch the exceptions.

Q29 (a), (b), and (c)

First, note that nested packages or nested static members are not automatically imported.

In (d), `p2.DefenceInDepth` is not a static member and therefore cannot be imported statically.

With (e), `March.LEFT` becomes ambiguous because both the second and the third import statements statically import `March`. The enum constant `LEFT` cannot be resolved either, as its enum type `March` cannot be resolved.

With (f), the enum constant `LEFT` cannot be resolved, as none of the static import statements specify it.

The enum type `p2.March` is also not visible outside the package.

Q30 (b)

Statement (c) is `false`, since an object of `B` can be created using the implicit default constructor of the class. `B` has a default constructor since no constructor has been defined. Statement (d) is `false`, since the second constructor of `C` will call the first constructor of `C`.

Q31 (c)

The important thing to remember is that an instance of a class is also an instance of its superclasses in the inheritance hierarchy.

Q32 (d)

Enum constants can be used as `case` labels and are not qualified with the enum type name in the `case` label declaration. The `switch` selector expression is compatible with the `case` labels, as the reference `this` will refer to objects of the enum type `Scale5`, which is the type of the `case` labels. The call to the method `getGrade()` returns a `char` value, which in this case is `'C'`.

Q33 (b) and (g)

The method in (a) and the method at (1) do not have the same or covariant return types required for overriding.

The method in (b) overrides the method at (2).

The instance method in (c) cannot override the static method at (4).

The static method in (d) and the static method at (4) do not have compatible return types for overriding.

The static method in (e) cannot hide the instance method at (3).

The instance method in (f) and the instance method at (5) do not have compatible return types for overriding.

The instance method in (g) overrides the instance method at (6), and they have covariant return types.

Q34 (c)

A primitive value cannot be widened and then boxed implicitly. The primitive value is boxed to its corresponding wrapper type, and an attempt is made to find a corresponding formal parameter with the most specific type to which it can be passed. The varargs value is passed in the method calls as follows:

Click here to view code image

```
printFirst(10);           // new Integer[] {Integer.valueOf(10)}
printFirst((byte)20);     // new Number[] {Byte.valueOf(20)}
printFirst('3', '0');     // new Object[] {Character.valueOf('3'),
                          //               Character.valueOf('0')}
printFirst("40");         // new Object[] {"40"}
printFirst((short)50, 55);// new Number[] {Short.valueOf(50),
                          //               Integer.valueOf(55)}
printFirst((Number[])new Integer[] {70, 75}); // Passed as array of Number
```

Q35 *(e)*

The program will compile and print `1`, `3`, and `2` at runtime. First, the static initializers are executed when the class is initialized, printing `1` and `3`. When the object is created and initialized, the instance initializer block is executed, printing `2`.

Q36 *(a)*

As there is no appropriate resource bundle file for the `no_NO` locale, the resource bundle for the default locale (US) is loaded. The value of the key `"greeting"` from this resource bundle is printed. Changing the default locale of the application does not change the locale associated with the resource bundle `rbs`. The value of the key `"greeting"` from this resource bundle is printed one more time.

Q37 *(e)*

The `runner` thread can only proceed if `intArray[0]` is not `0`. If this element is not `0`, it has been initialized to `10` by the main thread. If this element is `0`, the `runner` thread is put into the wait set of the `intArray` object, and must wait for notification. The main thread only notifies after initializing both elements of the array to `10`. Calling the `notify()` method on an object with no threads in its wait set does not pose any problems. A thread can only call `notify()` on an object whose lock it holds. Therefore, the last `synchronized` statement in the `main()` method is necessary.

Q38 *(b)*

Notice that the array is declared as a raw type, yet the objects that are placed into this array are of a parameterized type. This is a legal assignment because an object of a parameterized type can be assigned to a raw type variable, in this case an element within the raw type array. However, in this case the `compareTo()` method accepts an argument of type parameter `T`, which allows invocation of any `Object` method using a reference of this type parameter. The code in the `compareTo()` method invokes the `toString()` method to convert each value into a `String` object and then compare these strings. The order of the comparison is therefore not numeric, but lexicographical. Note that the raw type `Thingy` is also used in the `for(:)` loop.

Q39 *(c)*

Each lambda expression implements a `Comparable` that returns an `int` value. These `int` values are accumulated and stored in the static field `result` of class `Widget`. The first lambda expression returns the length of the `"ACME"` string (i.e., 4), the second lambda expression returns the index of letter `"C"` within the string `“ACME"` (i.e., 1), and the third expression returns the day of the month in the `LocalDate` object (i.e., `20`). The sum in the `result` field is thus `25`, which is printed.

Q40 (b)

The `sort()` method of the `Arrays` class sorts the elements of the `Integer` array according to the total ordering defined by the `Comparator<Integer>` that is implemented by the lambda expression. The difference `x - y` between two values `x` and `y` determines whether `x` is less than, equal to, or greater than `y`, according to the contract of the `compare()` method of the `Comparator<E>` interface. The elements of the array are sorted in ascending order.

Q41 (a)

The `LocalDate` class uses the ISO-8601 standard, where year 0 corresponds to 1 BC. The pattern `"dd MM yy G"` results in the number of the day in the month, the short name of the month, the two-digit value of the year, and BC/AD being used to format the local date.

Q42 (c)

Stream processing distinguishes `LocalDateTime` objects from all other objects. It extracts the number of seconds from a `LocalDateTime` object, substitutes 1 for all other elements in the stream, and calculates the sum for all the elements. There is only one `LocalDateTime` object in this stream, which yields the value of `2`; the other four objects are substituted with a value of `1` in the stream. Thus the result is `6`.

Q43 (d)

The text block has two incidental leading whitespace on each line that are removed. After the removal of the two incidental spaces from all lines, the line lengths in the text block are 1, 3, and 1. The resulting string literal is `"a\n b\nc\n"`.

Q44 (c)

The paths constructed by the code are as follows:

Click here to view code image

```
p1: /users/joe
p2: /users/bob
p1.relativize(p2):                  ../bob
p3 = p1.resolve(p1.relativize(p2)): /users/joe/../bob
```

```
p4 = p3.normalize():                  /users/bob
```

Since the first component of a path has index 0, index 1 refers to `joe` in `p3` and `bob` in `p4`.

Q45 (b)

The resource bundles are loaded from the files `resources_en.properties` (which has an entry for key `f2`) and `resources.properties` (which has an entry for key `f1`). The `resources_en_GB` bundle is not loaded by the `getBundle()` method.

When the value for key `f1` is not present in the `resources_en` bundle, the parent bundle `resources` is searched for the value of key `f1`. The `en_GB` locale is applied to the formatter, which uses the value `"yy-MMM-dd"` of key `f1` to format the date.

Q46 (b)

The `action()` method uses short-circuit conditional operators `||` and `&&` to test whether the parameter `obj` is a `String` or an `Integer`. The `instanceof` pattern match operator only introduces a pattern variable if it evaluates to `true`. If it is a `String`, it checks whether this string contains the character `'1'`. Otherwise, it checks whether the parameter `obj` is an `Integer`, and if it is, it determines whether its value is less than 1. The object passed to the method is an `Integer` object with a value of 1, which is not less than 1, and thus the `boolean` expression returns `false`.

Q47 (c)

A non-canonical constructor is required, since the arguments passed in the constructor are of type `String` and `int`, and not `String` and `Duration`, as in the declaration. The first statement in a non-canonical constructor must be an explicit invocation of a constructor with the `this()` expression that leads to the canonical constructor being invoked so that the component fields `title` and `duration` are initialized.

(a) and (b) are incorrect because they do not provide a value for the `title` field. Moreover, in (a), the `this` reference cannot be used in a `this()` expression.

(d) and (e) do not invoke the canonical record constructor with the `this()` expression.

(c) fulfills all the requirements.

Q48 *(b)*

The `walk()` method navigates the directory structure for the directory `./Sun`. The `map()` method extracts the name of the leaf element from each `Path` that is encountered in this depth-first walk in the directory tree. The names are sorted. Notice that although the names start with a character that represents a number, the natural ordering is that for strings.

The `limit()` method limits the length of the stream to just the first three elements. Given the natural ordering for strings, the first three elements in the list are `1_Io`, `1_Mercury`, and `1_Moon`. The `forEach()` operation prints a substring extracted from the name starting at index 2.

Q49 *(b)*

The method `lines()` creates a stream of text lines read from the text file. Lines are grouped on the substring extracted from each line after the `':'` character. The map created by grouping has these substrings as keys and the lines containing the key as values. All lines that end with 1.70 will be grouped together, all lines that end with 1.99 will be in another group, and so on.

Q50 *(c)*

Module `basic` is the service provider as it implements the service defined by the service interface `TransportType` in module `transport`. Module `basic` requires module `transport` and must declare that it provides an implementation (`basic.mode.Horse`) of the service interface `transport.mode.TransportType`.

The `requires` directive must specify modules, not packages, as in (b) and (d). (a) requires the wrong service module and implements the wrong service interface.

Appendix G

Java Logging API Overview

The Java Logging API is a supplementary topic, and *not* an exam objective. However, there could be some exam questions that utilize the Logging API to facilitate non-blocking parallel processing, such as in parallel streams. Also, an understanding of the Logging API principals and capabilities is an important practical programming skill.

G.1 Purpose of the Logging API

The purpose of the Java Logging API is to provide a simple yet flexible mechanism for writing logs to different destinations, while minimizing the effects of writing these logs on the overall performance of a Java application. A program can write messages to the standard output or standard error using `System.out` or `System.err`, respectively. However, this approach is not particularly flexible, because such write operations block the current thread. Thus they adversely affect program performance, and are totally unsuitable in any concurrent or parallel execution of code—for example, in parallel stream processing. Using standard out and standard error is also not very flexible because they do not allow a programmer to reconfigure the target destination for the log messages, nor do they provide a flexible dynamic mechanism to filter or format log messages.

The Logging API addresses the following issues:

- Provides a *configuration properties file* to configure:

    - *Log handlers*, which write log messages to different destinations
    - *Log message format* for each destination
    - *Logging levels*, which control how detailed logging should be done

- Minimizes performance implications with:
    - *Non-blocking* log writing operations
    - *Guarded logging*, which avoids unnecessary message formatting when the logger level is set in a way that a particular message is not going to reach the actual log destination

G.2 Configuring Logging

By default, the file `logging.properties` is used to configure Java logging. It is located in the `JAVA_HOME/conf` folder. (For Java version 8 or earlier, this was `JAVA_HOME/jre/lib` folder.) A different configuration file can be specified using the `java.util.logging.config.file system property`. For example:

Click here to view code image

```
java -Djava.util.logging.config.file=another-config
```

Log handlers are classes that are responsible for writing log messages to different destinations. The package `java.util.logging` provides a number of handler classes. A *logger configuration file* can be used to enable one or more handlers specified as a comma-separated list. By default, the logger configuration only sets up the *console handler*, but other handlers can be enabled as required. For example:

Click here to view code image

```
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

Each handler has its own specific configurable properties. For example, a file handler allows setting up a naming pattern for log file names, minimum size of the file, log rollover to the next file, and so on. For example:

Click here to view code image

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit   = 50000
java.util.logging.FileHandler.count   = 1
```

Obviously, such properties are not applicable to other types of handlers, such as the console handler. However, all handlers provided by the Java Logging API allow a logger format to be set; for example, as a simple plain text or xml format.

Click here to view code image

```
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.formatter    = java.util.logging.XMLFormatter
```

It is possible to implement customized handlers by extending the `java.util.logging.Handler` class or one of its subclasses: `MemoryHandler` or `StreamHandler`. Also, it is possible to implement a customized formatter by extending the `java.util.logging.Formatter` class.

Level configuration is used to control the number of messages that are actually written to a log destination. *Logging levels* are defined by the `java.util.logging.Level` class using the following constants:

```
SEVERE (highest value)
WARNING
INFO
CONFIG
FINE
FINER
FINEST (lowest value)
```

In addition, there is a level `OFF` that can be used to turn off logging, and a level `ALL` that can be used to enable logging of all messages.

When logging is set to a certain logging level, any log message that is associated with this logging level and levels above it will be written to the log destination. However, messages that are associated with any logging levels below the one that was set in the configuration will essentially be ignored by the logger.

For example, when the log level is set to `INFO`, the `INFO`, `SEVERE`, and `WARNING` level messages will be written to the log, but the `CONFIG`, `FINE`, `FINER`, and `FINEST` level messages will be ignored by the logger.

It is possible to implement additional logging levels by subclassing the `java.util.logging.Level` class.

Log level configuration defines a *global log level property*, but also allows overriding the level for specific loggers. For example:

```
.level=INFO
org.foo.level=SEVERE
```

It is important to note that by convention, logger names are essentially Java packages and class names. So this example implies that everywhere in the program, the global logger level is `INFO`, except for the package `org.foo`, where it is set to `SEVERE`.

It is also possible to set the logging level for a specific handler. For example:

Click here to view code image

```
java.util.logging.ConsoleHandler.level = WARNING
```

Suppose that logging has been configured to use a file logger and the console logger, and the console logger is restricted to the `WARNING` level, but no such restriction is imposed on the file logger. Also, assume that the global log level is set to `INFO`. This would mean that log messages associated with a level below `INFO` would all be ignored; only `WARNING` and `SEVERE` level message would be written to the console logger; and `INFO`, `WARNING`, and `SEVERE` level messages would be written to the file logger.

G.3 Writing Log Messages

To set up logging for a program requires a few basic steps. First, a logger object must be initialized, using the `java.util.logging.Logger.getLogger(String name)` method. By convention, the name of a logger object should correspond to the fully qualified class name from which the log is written. This way, it will match the logger naming convention used by the logger configuration file. For example:

Click here to view code image

```
package org.foo;
import java.util.logging.*;
public class Test {
  private static Logger logger = Logger.getLogger(org.foo.Test.class.getName()
  public static void main(String[] args) {
  ...
  }
}
```

Notice that the logger name could theoretically be any string.

However, using package and class names really helps to organize logger configuration. In this case, a logger config file can associate the `org.foo` logger name with a particular level of logging, which is going to affect log messages written from the `org.foo.Test` class.

Once the logger object is initialized, it can be used to write messages, using the `logger.log()` method. Each message is associated with a specific logger level and may contain any text. The `log()` method is overloaded to allow the program to pass an exception object as an additional argument so that exception details can be passed to the logger as part of the message. For example:

Click here to view code image

```
package org.foo;
import java.util.logging.*;
public class Test {
  private static Logger logger = Logger.getLogger(org.foo.Test.class.getName()
  public static void main(String[] args) {
    try {
      // any actions that may potentially throw an exception
    }catch(Exception e) {
      logger.log(Level.ERROR, "Some error message", e);
    }
    logger.log(Level.INFO, "Some other message");
  }
}
```

The `Logger` class provides a number of additional convenience methods that are equivalent to the `log()` method:

> `logp()`
>
> The *log precise* method takes additional parameters that specify a class and a method name.
>
> `logrb()`
>
> The *log with resource bundle* method takes a resource bundle name that may be used to localize log messages.
>
> `entering()`, `exiting()`, `throwing()`
>
> These are convenience methods used to log method entry, method exit, and any exception thrown at the `FINER` level.
>
> `severe()`, `warning()`, `config()`, `info()`, `fine()`, `finer()`, `finest()`
>
> These are convenience methods, each hardcoded to use a specific log level, rather than indicating the log level as an argument. In other words, `logger.info("My message")` is equivalent to `logger.log(Level.INFO, "My message")`.

All of these log methods are *non-blocking*. This means that when any of these log methods are invoked in the program, the current thread is not blocked, as the method call returns immediately. The logger accumulates messages produced by the different program threads, and writes these messages to the actual destinations in another thread of its own. As a result, the log messages can reach their destinations with a minimum of delay, but the performance impact on the overall program execution is minimized. This allows loggers to be utilized in concurrent and parallel execution scenarios.

G.4 Applying Guarded Logging

Finally, it is a good idea not to format log messages ahead of time. This is because it is not known whether the message would or would not be written into a log because of the possible changes to the logger configuration. This means that code in the following example is not advisable:

Click here to view code image

```
logger.log(Level.FINE, "This is " + someVar + " value");
```

In the preceding code, message concatenation is going to be performed even if the log level could be set in a way that would ignore this message. In order to prevent such unnecessary processing, it is advised to use *guarded logging*. Basically, the aim is not to do message formatting when the message is going to be discarded because of the log level settings. This could be achieved with an `if` statement or by using *message value substitution*. For example, the following `if` statement:

Click here to view code image

```
if(logger.isLoggable(Level.FINER)) {                         // (1)
  logger.log(Level.FINE, "This is " + someVar + " value");
}
```

can be witten using message value substitution:

Click here to view code image

```
logger.log(Level.FINE, "This is {0} value", someVar); // (2)
```

Notice that the code at (2) looks much cleaner than the code at (1) and is thus probably the best way to handle log messages.

G.5 Summary

The Logging API provides a simple, yet flexible non-blocking mechanism to write information about program execution, including general information as well as error messages. The certification exam does not ask any direct questions about logging, but use of loggers may be encountered in the context of other questions, specifically where non-blocking output is important, such as in the context of the Concurrency API or execution of parallel streams.

Index

Symbols

`-` 63
`--` 69
`^` 78, 82
`^=` 79
`_` 36, 1103
`,` 1125
`;` 101
`:` 157, 340, 1100, 1287
`::` 724
`!` 78, 1101
`!=` 75, 426
`?` 580, 582, 588
`? extends T` 581, 589
`? super T` 582, 589
`?:` 90
`.` 9, 127, 128, 326, 338, 1125, 1288
`..` 1289
`...` 136, 141 `'` 37, 38
`"` 39
`"""` 458
`[]` 92, 118, 120
`{}` 101, 119, 354
`@` 1558, 1563
`@Deprecated` 1580

@Documented 1574

@FunctionalInterface 1579

@Inherited 1574

@Override 198, 224, 241, 618, 1578

@param 100, 112

@Repeatable 1575

@Retention 1567

@return 184

@SafeVarargs 632, 650, 659, 863, 1585

@SuppressWarnings 1582

@SuppressWarnings("unchecked") 627, 632

@Target 1569

@throws 390

* 58, 61, 330

*= 66

/ 61, 62, 1287

/* and

*/ 40

/** and

*/ 41

// 40

/= 66

\ 38, 1101, 1287

& 78, 82

&& 80

&= 79

# 1101, 1125, 1146

% 61, 62

%= 66

+ 63, 67

+ concatenation 448

++ 69

+= 66

< 74, 1146

<< 86, 87

<<= 90

<= 74

<> 567, 570, 646

-= 66

= 54, 1100

== 75, 426, 433

-> 92, 160, 679

> 74

>= 74

>> 86, 88

>>= 90

>>> 86, 89

>>>= 90

| 78, 82, 399, 1146

|= 79

|| 80

~ 82

Numerics

0 1125

A

ability interfaces

see marker interfaces

abrupt method completion 367

absolute adjusters 1035

absolute path 1288

absolute values 478

abstract

  - interfaces 239
  - methods 240, 675

`abstract` 239, 240, 675

`abstract` classes 218

  - `abstract` methods 224
  - considered incomplete 222
  - declaring 219
  - extending 220
  - incomplete 219
  - polymorphic references 220
  - UML class diagram 219

abstract data type 565

`abstract` keyword 219, 224

abstract method declarations 675

  - in enum types 294
  - in interfaces 240

`abstract` methods 224

  - declaration 224
  - implementing 224
  - method header 224
  - overloading 225
  - overriding 224

abstraction 5, 13

access modifiers 99, 100, 109, 345

access order 841, 842

accessibility 10, 19, 352, 1173

  - access modifiers 345
  - anonymous classes 525
  - local classes 512, 513
  - members 347, 352

- modifiers 493
- non-static member classes 504
- static member classes 498
- top-level types 345
- UML notation 348

accessibility for class members

- package 347, 351
- `private` 351
- `protected` 350
- `public` 350

accessibility for top-level types

- package 345
- `public` 345

accessibility rules 1174
accessor methods *see* get methods
accounting currency formatting 1118
acquisition order 1461
activation frame 534

- *see* method execution 365

actual parameter 128
actual parameter list 127, 265
actual type parameters 569
adapter 992, 993, 994
adding to class 191
additive operators 63
aggregate module 1168
aggregate of values 300
aggregate operations 881
aggregation 13, 16, 194

- hierarchy 195

algorithms 783
aliases 55, 76, 130

  see also symbolic link

all-or-none model 1474
ambiguous call 202, 266
analyzing program code 1620
and operator 83
AND-ing 86
`AnnotatedElement` 1588
`Annotation` 1557
annotation 1557

  `@` 1563
  applying annotation type 1563
  declaration annotation 1571
  default values 1565
  marker annotation 1563
  metadata 1557
  normal annotation 1563
  processing 1587
  single-element annotation 1563
  type annotations 1572

annotation type 1557, 1558

  `@` 1558
  access modifier 1559
  annotation type element 1560
  containing annotation type 1576
  declaring 1558
  marker annotation type 1560
  member declarations 1559
  multi-element annotation type 1561
  repeatable annotation type 1576

- simple name 1559
- single-element annotation type 1560
- syntax 1558

annotation type declaration

- *see* annotation type

annotation type element 1560

- access modifier 1561
- attributes 1560
- default values 1562
- element type 1561
- method declaration 1560
- optional annotation element 1562
- required annotation element 1562
- `value()` element 1560, 1561

annotations 239

- `@Deprecated` 1580
- `@FunctionalInterface` 675, 1579
- `@Override` 198, 618
- `@SafeVarargs` 632
- `@SuppressWarnings("unchecked")` 627

anonymous arrays 122, 126

- `[]` 122

anonymous classes 494, 521, 571, 690

- accessing members in enclosing context 525
- compared with lambda expressions 692
- declaration 521
- event listeners 521
- extending an existing class 521
- hiding members in enclosing context 525

  - implementing an interface 523
  - in non-static context 521
  - in static context 521
  - instantiation 524
  - objects on the fly 521
  - referencing instances 524
  - shadowing local variables 525, 692
  - using qualified `this` 525, 693

- anti-symmetric 762
- API (application programming interface) 2
- apostrophe 38
- application 19
- architecture neutral 4
- argument

  - *see* actual parameter

- arguments to main method 141
- arithmetic compound assignment operators 66
- `ArithmeticException` 372
- arity 136, 695
- array constructor references 725, 731
- array creation expression 119, 122, 143
- array element initializer 1564
- array initializer 119, 120, 122, 126, 146
- array list 802
- array mismatch 871
- array operations 798
- array store check 260, 627
- array store check report

  - false positive 631

- array types 579

- *see* arrays

`ArrayBlockingQueue` 1496

`ArrayDeque` 823

`ArrayIndexOutOfBoundsException` 121, 372, 397

`ArrayList` 450, 802

  - add collection 651
  - add element 647, 651
  - autoboxing 654
  - capacity 646
  - clear list 652
  - comparison with arrays 662
  - constructing 646
  - constructors 648
  - converting to array 658
  - element search 656
  - filtering 688
  - import 646
  - inheritance hierarchy 645
  - initial capacity 647
  - insertion order 644
  - list of lists 648
  - membership test 655
  - modifying 651
  - nested lists 648
  - object value equality 656
  - open range-view operations 645
  - ordered 644
  - position-based access 645
  - positional index 655
  - positional order 644
  - positional retrieve 655
  - querying 655

  - references 646
  - remove collection 652
  - remove element 652
  - replace element 652
  - size 646, 655
  - subtype relationship 648
  - text representation 647
  - trim to size 653
  - type-safety 646, 648
  - unchecked conversion warning 646
  - zero-based index 644

`Arrays` 868

  - creating streams 898

arrays 117, 425

  - `[]` 118, 119, 120
  - `{}` 119
  - anonymous 122, 126
  - array creation expression 119
  - array initialize list 119, 122
  - array initializer 120, 126
  - array name 118
  - array size 119
  - array store check 260, 627
  - `ArrayIndexOutOfBoundsException` 121
  - binary search 866
  - bounds 121
  - collection operations 798
  - common prefix 870, 871
  - comparators 867
  - comparing 868
  - construction 119

- converting to list 659
- declarations 118
- default initialization 118, 119
- dynamic 645, 802
- element access expression 120
- element default value 260
- element type 118
- elements 117, 120
- equality 868
- filling 873
- index 117
- index expression 120
- initialization 119, 124
- initializing 873
- iterating over 176
- iterating over an array 121
- length 118
- lexicographic comparison 869
- mismatch 871
- multidimensional 122, 124
- objects 425
- proper prefix 871, 872
- ragged 124
- reference 118, 121, 260
- searching 866
- sequential streams 898
- sorting 865
- subtype covariance 259
- text representation 873
- using 120

arrays of arrays 118, 124

- multidimensional 124

`ArrayStoreException` 260, 372, 648, 658, 798, 799
array-valued single-element annotation 1566
arrow notation (->) 160
arrow operator (->) 92
ascending-order key set 847
ASCII 37, 42
`AssertionError` 374
assessor methods
    see get methods
assignable 47, 264
assignment
    widening reference 584
assignment compatible 47, 264, 646
assignment conversions 47
assignment operator 8
assignments
    arithmetic compound operators 66
    bitwise 85
    cascading 55
    compound operators 79, 85
    expression statement 55
    extended operator 90
    implicit narrowing 56
    multiple 55
    numeric conversions 56
    operator 50, 54
    primitive values 54
    references 55
    widening reference conversions 196
association 16

    aggregation 194
    composition 195
    realization 244

associativity 51, 52
asynchronous 1370
atomic actions 1455, 1456, 1469

    CAS: Compare and Set 1457

atomic classes 1457
atomic move 1308
atomic operation 1305, 1339
atomic variables 1456

    atomic actions 1456

`AtomicInteger` 1457, 1458
atomicity 1455
`AtomicMoveNotSupportedException` 1313
at-sign 1558
attributes 1512
attributes *see* properties
`AttributeView` interface 1334
autoboxing 430

    `for(:)` statement 177
    see boxing conversion

`AutoCloseable` 409, 412, 536, 1234, 1244, 1519

    streams 889

automatic garbage collection 533
automatic memory management 533
automatic module 1206, 1207
naming 1207
Automatic Resource Management (ARM) 411

automatic variables *see* local variables
auxiliary methods 252
available time zones 1073

B

backing collection 856
backslash 38
backspace 38
backtrace 405
backward compatibility 565
bags 804
balanced tree 788, 789, 812
barrier
    see cyclic barrier
barrier action 1471
barrier point 1471
base 33, 431, 435
base case 1448
base class 191
base name 1102
basic collection operations 790
basic `for` statement 174
basic map operations 831
basic statement 1523
`BasicFileAttributes` interface 1330
`BasicFileAttributeView` interface 1334
Before Current Era (BCE) 1029
behavior 675
behavior parameterization 675
behavioral parameters 909
`BiConsumer` functional interface 709

  - primitive type specializations 711

`BiFunction` interface 717, 718

  - composing two-arity functions 718
  - primitive type specializations 719

big numbers 484, 485

`BigDecimal`

  - customizing formatting 1125
  - formatting 1117
  - parsing 1119
  - rounding 1122

`BigDecimal` class 485

`BigInteger` 485

binary

  - numeric promotion 50
  - operators 50
  - shift operators 86

binary files 1238, 1239, 1240

binary operator 52

binary representation

  - *see* binary values

binary search

  - arrays 866
  - lists 861

binary values 1238, 1239, 1240

`BinaryOperator` interface 721, 722

  - composing binary operators 721
  - primitive type specializations 722

bind parameter 1526

bind variable 1526
binding of method call 205, 227
`BiPredicate` functional interface 703
bit mask 84
bit masking 86
bit patterns 54
bit shifting 86
bitwise
  - *and* operator 83
  - assignment 85
  - complement 83
  - compound assignment 85
  - operators 82
  - *or* operator 83
  - shifting 86
  - *xor* 83

bitwise AND
  - `&` 82

bitwise complement
  - `~` 82

bitwise exclusive OR
  - `^` 82

bitwise OR
  - `|` 82

bitwise XOR
  - `^` 82

blank `final` local variables 232
blank `final` variables 135, 229, 231

block 161, 166, 169
block scope 169, 355, 684
`BLOCKED` state 1382
blocking collection 1483
blocking operations 1234, 1405
blocking queues 1495
`BlockingDeque` 1496, 1498
`BlockingQueue` 1496, 1497
blocks 101, 354
    scope 354
    `try` 377
boilerplate code 409, 691
`Boolean`
    condition 152
`Boolean` wrapper class 437
booleans 41, 43
    casting 49
    expressions 74
    literals 37
`BooleanSupplier` functional interface 699
bounded collection 1483
bounded instance method references 727
bounded type parameters
    constraint 591
bounded wildcard 581
bounds
    constraint 591
boxing conversions 45
`break` statement 157, 180

bridge method 615, 764
BS *see* backspace
buckets 753
buffered readers 1251

  creating streams 1252
  end of stream 1252
  reading text lines 1252

buffered writers 1250
`BufferedReader` class 1242, 1315

  streams 902

`BufferedWriter` class 1243, 1316
buffering streams 1238
building abstractions 13
built-in functional interfaces

  `java.util.function` package 695
  `BiConsumer` 709
  `BiFunction` 717, 718
  `BinaryOperator` 721, 722
  `BiPredicate` 703
  `BooleanSupplier` 699
  `Consumer` 709
  `DoubleBinaryOperator` 722
  `DoubleConsumer` 709
  `DoubleFunction` 712
  `DoublePredicate` 703
  `DoubleSupplier` 699
  `DoubleToIntFunction` 713
  `DoubleToLongFunction` 713
  `DoubleUnaryOperator` 720
  `Function` 712
  `IntBinaryOperator` 722

  - `IntConsumer` 709
  - `IntFunction` 712
  - `IntPredicate` 703
  - `IntSupplier` 699
  - `IntToDoubleFunction` 713
  - `IntToLongFunction` 712
  - `IntUnaryOperator` 720
  - `LongBinaryOperator` 722
  - `LongConsumer` 709
  - `LongFunction` 712
  - `LongPredicate` 703
  - `LongSupplier` 699
  - `LongToDoubleFunction` 713
  - `LongToIntFunction` 713
  - `LongUnaryOperator` 720
  - `ObjDoubleConsumer` 709
  - `ObjIntConsumer` 709
  - `ObjLongConsumer` 709
  - `Predicate` 703
  - `Supplier` 699
  - `ToDoubleBiFunction` 718
  - `ToDoubleFunction` 712
  - `ToIntBiFunction` 718
  - `ToIntFunction` 712
  - `ToLongBiFunction` 718
  - `ToLongFunction` 712
  - `UnaryOperator` 720

- built-in synchronization
  - *see* intrinsic locking

- bulk map operations 839
- bulk operations on collections 790

bulk task execution 1439
bundling resources 1102

  `byte` 32, 42

byte streams 1233

  reading bytes 1234
  writing bytes 1234

bytecode 4, 21
bytecode verification 1611

C

C++ 4
cached thread pool 1426
caching 358
call by reference 131
call by value 131
call chaining 128
call signature 266
call stack

  see JVM stack 365

callable statement 1530

  stored functions 1530
  stored procedures 1530

callee 127
caller 127, 184
candidate bundle name 1108
capacity 646
capture conversion 606
capture of 605
carriage return 38, 39

CAS: Compare and Set 1457
cascaded grouping 987
cascaded partitioning 991
cascading assignments 55
cascading `if-else` statements 155
`case` constants 156, 157, 160
`case` labels 156, 157, 160
case sensitivity 30
case-insensitive ordering 771
cast 631
  references 281
cast operator 45, 48, 50, 57, 66, 67, 76, 143, 269
casting 47, 48
  see also conversions
`catch` clause 377
  multi- 397
  rethrowing exception 401
  uni- 376
catching exceptions 365
catch-or-declare rule 388
catch-or-specify requirement 388
cause 405
ceiling entry 846
ceiling key 846
chaining 555
  constructors 213, 555
  `new` operator 507
chaining exceptions 405
changing list contents 863

  - `char` 42

character case 448

character encoding 1241
  - *see also* character sets

character sequences
  - *see* strings *and* string builders

character set
  - ASCII 37, 42, 1241
  - ISO Latin-1 37, 42
  - Unicode 37, 42, 1241

character streams 1233, 1241
  - character encoding 1241
  - end of stream 1243
  - reading characters 1243
  - writing characters 1244

`Character` wrapper class 437

character-based device
  - console 1256

characters 42
  - literals 37
  - searching for 451, 466

`CharSequence` interface 444, 450, 901

charsets
  - *see also* character sets

checked exceptions 374

child class 191

child threads 1373, 1378, 1454

choice format 1146, 1148

- limits 1146
- subformat pattern 1146
- syntax 1146

`ChoiceFormat` 1115, 1148
choosing between `String` and `StringBuilder` class 464
`ChronoField` enum type 1036, 1046, 1079

- list of enum constants 1046

`ChronoUnit` enum type 1040, 1041, 1044, 1081, 1082

- list of enum constants 1044

`Class` 1588
`Class` class 426
class design 1611
class file 21
class file name 497
class hierarchy

- *see* inheritance hierarchy

class initialization 230
class literals 636
class lock 1380, 1391, 1392
class method 13
class path 337, 1206

- absolute pathnames 340
- entries order 340
- entry-separator character 340
- fully qualified package name 340
- path-separator character 340
- relative pathnames 340
- search in a named package 339

  - searching for classes 337
  - whitespace 340

class search path

  - *see* class path

class variable 13

class versioning 1273

`ClassCastException` 271, 372, 1105

classes

  - `abstract` 218
  - accessibility 345
  - adding to 191
  - anonymous 492, 494, 521
  - base 191
  - body 99
  - child 191
  - class file name 497
  - concrete 218, 220, 226
  - constructors 109
  - contract 218, 242
  - declaration context 493
  - declarations 99, 325
  - definitions 6, 8
  - derived 191
  - diagram 11, 12
  - encapsulation 324
  - extending 191, 225
  - `final` 225, 358
  - `final` vs. `abstract` 226
  - fully qualified name 337
  - fully qualified package name 328
  - generalized 194

grouping 326
header 99
immutable 356
implementation 218
implementing interfaces 240
incomplete 219
initialization 230, 557
inner *see* inner classes
instances 8
local 492, 494, 512
members 9
methods 115
name 326
nested *see* nested classes
non-static member 492, 494, 501
`Object` 425
on-the-fly 492
parent 191
partial implementation 218
records 299
runtime 426
scope 352
sealed 312
searching for 337
specialized 194
static fields 113
static members 113, 494, 495
static methods 115
static variables 113
subclass 13, 191
superclass 13, 191
top-level 491, 494
utility 115

- wrappers 425, 429

class-level lock

- *see* class lock

`ClassLoader` class 425

`ClassNotFoundException` 371

`CLASSPATH` environment variable

- *see* class path

`-classpath` option

- *see* class path

clauses

- `catch` 377
- `extends` 191
- `finally` 377, 382
- `implements` 240
- `permits` 312
- `throws` 388

cleaning up 536

clean-up code 382, 1378

client 10

`Cloneable` interface 427

`CloneNotSupportedException` 427

cloning objects 427

`Closeable` 412

closing an I/O stream 1234, 1239, 1240

closing delimiter 458

code corruption 1606

code injection 1605

code migration

- bottom-up strategy 1209

  - top-down strategy 1210

code optimizations 229

Code Ranch (`www.coderanch.com`) 1616

code reuse 3, 191, 195

codebase 1608

`Collection` 644

`Collection` interface 784

collections 586, 644, 783, 784

  - adding elements 863
  - array operations 798
  - `ArrayList` 802
  - as single entity 644
  - basic operations 790, 831
  - bulk operations 790, 839
  - `Cloneable` 787
  - compare with streams 888
  - data structures 788
  - duplicate elements 801, 804
  - elements 644, 783
  - filtering 796
  - first element 810, 811, 1486
  - `HashSet` 806
  - implementations 783, 785, 788
  - interfaces 783
  - iteration 791, 792
  - iteration order 795
  - iterators 793
  - last element 810, 811, 1486
  - `LinkedHashSet` 806
  - `LinkedList` 802
  - `List` 802
  - lists 801

  - map views 839
  - maximum 862
  - minimum 862
  - `NavigableSet` 810
  - optional operations 790
  - ordered 644, 789, 801
  - parallel streams 897
  - range-views 801, 810, 811, 1486
  - retaining elements 791
  - sequential streams 897
  - `Serializable` 787
  - `Set` 804
  - shuffle elements 860
  - sorted 789
  - `SortedSet` 810, 811
  - sorting 644
  - subset 810, 811, 1486
  - text representation 795
  - thread-safety 802, 840
  - `TreeSet` 812, 847
  - unmodifiable view 856
  - unordered 789
  - unsupported operations 831
  - utility methods 783
  - `Vector` 802
  - views 656, 810, 811, 839, 1486

- collections framework 783
- `Collections` utility class 858, 861
- `Collector` interface 890, 978
- `Collectors` 890, 978
  - importing 978

collectors 948, 964, 978

- averaging 1000
- classification mapping 979
- classification maps 979
- classifier function 979
- collecting to a `Collection` 979
- collecting to a `ConcurrentMap` 983
- collecting to a `List` 980
- collecting to a `Map` 981
  - key mapper 981
  - map supplier 982
  - merge function 981
  - value mapper 981
- collecting to a `Set` 980
- comparison with terminal operations 1008
- composing 979
- concurrent 983, 989
- counting 998
- downstream 979, 985, 986, 992, 993, 994, 997, 998
- filtering 992
  - downstream collector 992
  - predicate 992
- finding min/max 999
- finishing 997
  - downstream collector 997
  - finisher function 997
- flat mapping 994
  - downstream collector 994
  - mapper function 994

  - functional reduction 998, 1002
  - grouping 979, 985
    - classification map 985
    - classifier function 985
    - default downstream collector 985
    - downstream collector 986
    - map supplier 986
    - multilevel 987
    - to a `ConcurrentMap` 989
  - joining to a string 984
  - mapping 993
    - downstream collector 993
    - mapper function 993
  - partitioning 979, 989
    - classification map 990
    - default downstream collector 990
    - downstream collector 990
    - multilevel 991
    - partitioning predicate 989
  - reducing 1002
    - binary operator 1002, 1003
    - identity value 1002, 1003
    - mapper function 1003
  - summarizing 1001
  - summary of 1005
  - summing 1000
- colon notation (:) 156
- columns 1512
- command line 21, 141

command prompt 21
commands
    `java` 21
    `javac` 20
comments 40
common memory space 1367
common prefix 870, 871
communication 127
compact canonical record constructor 304, 306
compact form styles 1120
compact number formatting 1120
compact number parsing 1121
compact strings 440
`CompactNumberFormat` 1115
`Comparable` interface 432, 447, 761
    `compareTo` 761
comparable objects 761
`Comparator` interface
    `compare` 769
    functional interface 769
comparators 761
    conditional 765, 771, 773
    reverse order 859
comparing objects 426, 744, 769
    see also equality
comparing strings 447
comparison 74
compilation unit 328

  deriving permitted direct types 317

compiler

  `-Xlint:unchecked` option 577

compiler optimizations 1453
compiling and running a service 1203
compiling Java source code 20
complement

  ~ 83

completes abruptly

  see exception propagation 367

component fields 302
component list 300
composite object 13
composition 16, 195

  see aggregation

compound declaration 146
compound messages

  conditional formatting 1145
  formatting 1143
  handling plurals 1146

compound mutually exclusive operations 1480
compound state 1382
compound statement 101
concatenation of strings 448
concatenation operator 67
concise `try`-with-resources 411
concrete class 218
concrete method 226

concrete parameterized types 580
concrete type 573, 580, 586
concurrency 1368, 1540
concurrent activities 1367
concurrent collections 1482, 1483

- blocking 1483
- bounded 1483
- duplicate elements 1483
- happens-before relationship 1483
- non-blocking 1483
- ordering 1484
- unbounded 1483

concurrent collector 983, 989
concurrent maps 983, 989, 1490

- segments 1491

`ConcurrentHashMap` 1491
`ConcurrentLinkedDeque` 1486, 1487
`ConcurrentLinkedQueue` 1486, 1487
`ConcurrentMap` 983, 989, 1491
`ConcurrentModificationException` 658, 795, 796
`ConcurrentNavigableMap` 1492
`ConcurrentSkipListMap` 1491
`ConcurrentSkipListSet` 1486
condition

- `Boolean` 152
- expressions 152

conditional 74

- operators 90
- statements 152

conditional *and* operator 80

  - pattern variable 277

conditional comparator 765, 771, 773
conditional expressions 90

  - associativity 92
  - nested 92
  - precedence 91
  - short-circuit evaluation 91
  - side effects 91

conditional formatting 1115, 1145
conditional operators 80
conditional *or* operator 80

  - pattern variable 277

conditions 74
configuration 5
configuration data 1100
congruent 762
connecting punctuation character 30
connection 1519
`Console` 1153
console 1256

  - formatted prompting 1257
  - printing formatted values 1257
  - reading password 1256

`Console` class 1256
`const` 32
constant 546
constant expression 47, 56, 69, 157, 229, 546, 1562

  - labels 157

constant field values

    `case` labels 162

constant string expressions 162
constant values 32, 229
constant variable 56, 229, 546
constants 254
constant-specific class bodies

    using anonymous classes 294

constituent objects 13
constituents 16
constraints 1512
constructing array 119
constructor chaining 210, 213, 555
constructor references 725, 730
constructors 7, 109

    access modifier 109
    accessibility 352
    body 109
    canonical 302
    chaining 210, 213
    class name 109
    declaration 99
    default 110
    header 109
    implicit default 110
    local declarations 109
    no-argument 110, 210, 213
    non-zero argument 111, 214
    overloading 112
    superclass constructor 110

constructs 30

  - high-level 30
  - loops *see* iteration statements
  - `synchronized` statements 1391, 1392

- `Consumer` functional interface 709
  - composing consumers 710
  - primitive type specializations 711
- contained annotation type 1562
- containers 586
  - *see also* collections
- containing annotation type 1562, 1576
- contains characters 452
- context switching 1368
- contextual keywords 31
  - `non-sealed` 312
  - `permits` 312
  - `sealed` 312, 315
  - `yield` 164
  - `continue` statement 182
- contract 6, 195, 218, 240, 324
- control flow
  - `break` 157, 180
  - `continue` 182
  - `do-while` 173
  - `for(;;)` 174
  - `for(:)` 176
  - `if` 152
  - `if-else` 153
  - iteration *see* iteration statements
  - loops *see* iteration statements
  - `return` 184

  - statements 101
  - `switch` expression 164
  - `switch` statement 155
  - `throw` 386
  - transfer statements 179
  - `while` 172
  - `yield` 164
- control transfer 179
- conversion categories 47
- conversion contexts 47
- conversions 261
  - assignment 47
  - contexts 46
  - explicit 43
  - identity 67
  - implicit 43
  - implicit narrowing 67
  - method invocation 48
  - narrowing reference 270, 585, 605
  - numeric promotions 49
  - parameters 129
  - reference casting 269
  - string concatenation 68
  - to strings 454
  - truncation 57
  - type-safe 264
  - unsafe casts 271
  - widening reference 196, 270, 584
- converting values 431, 432, 434, 435, 437, 454
- copy-on-write collections 1501
- `CopyOnWriteArrayList` 1502

`CopyOnWriteArraySet` 1502, 1503
`CopyOption` interface 1301
Core Collections Interfaces 786, 1485, 1495
core interfaces 783
Core Map Interfaces 787
core stream interfaces 890
count-down latch 1474
`CountDownLatch` 1474
counter-controlled loops 174
country codes 1097
covariant return 197, 201, 619
`-cp` option
    see class path
CPU core 1009
CR *see* carriage return
crab 176
`CREATE` statement 1514
creating
    objects 92
    threads 1370
creating a runtime image 1204
creating modular JAR 1189
creating records 303
critical section 1388
CRUD operations 1514
currency 1096, 1116
    currency symbol 1126
    formatting 1116, 1119
currency formatter 1117, 1119
currency symbol 30, 1126

current directory 1289

- . 338, 1288
- system property 1292

Current Era (CE) 1029

current object 106

currying functions 723

- partial application 723

cursor 801, 1534

cyclic barrier 1471

- all-or-none model 1474
- barrier action 1471
- barrier point 1471

`CyclicBarrier` 1471

D

`-d` option 335

daemon 1377, 1402

dangling references 533

Data Definition Language (DDL) 1514

data encryption 1604

data integrity 1606

Data Manipulation Language (DML) 1514

Data Query Language (DQL) 1514

data source 884

data streams 1236, 1238

data structures 644, 788

data types *see* types

database capabilities 1543

database connection 1519

database cursor 1534, 1540

Database Management System (DBMS) 1512
database protocol 1520
database transactions 1545
`DataFormat` 1115
`DataInputStream` class 1235
`DataOutputStream` class 1236
`Date` 1024, 1078, 1141
date 761, 1096

  see temporal objects

date fields 1024
date units 1040
`DateFormat` 1141
date-time

  see temporal objects

date-time formatters

  format styles 1132
  immutability 1128
  ISO-based default 1127, 1129
  ISO-based predefined 1127, 1129
  letter pattern 1134
  parsing 1128
  pattern letters 1134, 1135
  pattern-based 1134
  style-based 1127, 1131
  thread-safety 1128

`DateTimeException` 373, 1025, 1033, 1075
`DateTimeFormatter` 1115, 1127
`DateTimeParseException` 374, 1051, 1058, 1066, 1131, 1134
daylight saving time (DST) 1073, 1082

  time gap 1082

  - time overlap 1082
  - time zone rules 1082, 1084
  - vs. standard time 1082

`DayOfWeek` enum type 1033

DBMS 1512

DDL 1514

DDL operations 1514

deadlocks 1408

decimal separator 1125

`DecimalFormat` 1115, 1125

declaration 99

declaration annotations 1571

declaration context 493, 1571

declaration statement 8, 65, 70, 81, 102, 103, 176

declaration-before-reading rule 547

declarations
  - arrays 92, 118
  - classes 99, 325
  - interfaces 325
  - local 101
  - main method 141
  - methods 100
  - multidimensional arrays 122
  - packages 325, 328
  - statements 101
  - variable arity method 136

declare-before-use rule 572

declared type 203, 264
  - *see* static type

declared-type parameters 680

declaring *see* declarations

decoupling 283
decrement operator 69
deep copying 426
deep reflection 1191
default
    constructor 110
    exception handler 367
    values 103, 541, 555
`default` 246
    label 156, 161, 166
    method 246
default accessibility for class members
    see package accessibility for class members
default accessibility for top-level types
    see package accessibility for top-level types
default constructor 110
default downstream collectors 985
default file system 1289
default locale 1098, 1099, 1104, 1107, 1109, 1119
default method 246, 675, 676
    conflict resolution 248
    multiple inheritance of implementation 248
    overriding 247
default ordering
    see natural order
default package 328
default resource bundle 1102, 1103, 1109, 1110
default values 1562, 1565

defensive copying 359
deferred execution 679, 684
definite assignment analysis 105, 232
definitions

- interfaces 238
- polymorphism 278

`DelayQueue` 1496
`DELETE` statement 1516
delimiters 1120
Denial of Service (DoS) 1602
dependency cycles 1172
deprecated 429
deprecated code

- `@Deprecated` 1580
- deprecation warning 1581
- ordinarily deprecated 1580
- terminally deprecated 1580

deprecation warning 1581, 1583
`Deque` 821
deque 785, 788

- `ArrayDeque` 823
- `Deque` 821
- FIFO queue 821
- head 821
- `LinkedList` 823
- tail 821

Derby database 1520

- connection 1520

derived class 191
deserialization 1611

deserialization of objects 1261
destination directory 335, 1187, 1219
destination object 1152
destination of data 1233
destination stream 25
destructive collection operations 791
diagrams

- class 6
- object 9
- *see also* UML

diamond operator (<>) 570
dictionary 785
dictionary order 447
difference 804
direct sealed superclass 313
direct superclass 211
directory

- copy entire directory 1352
- *see also* directory entry

directory entry 1288

- accessing individual file attributes 1321
- atomic move 1313
- copy entire directory 1352
- copy options 1308
- copying 1308
- copying using I/O streams 1311
- creating 1339
- creating regular directories 1342, 1343
- creating regular files 1339, 1340
- creating streams 1345
- creating symbolic links 1342

- creating temporary directories 1342, 1344
- creating temporary files 1339, 1341
- creation time 1324
- deleting 1307
- denotes same as another entry 1306
- determining existence 1305
- determining file accessibility 1324
- determining kind of entry 1323
- determining last modification time 1324
- determining owner 1325
- determining size 1323
- handling POSIX file permissions 1325
- hidden 1324
- last access time 1324
- last modified time 1324
- managing file attributes 1321
- moving and renaming 1312
- reading and writing bytes 1318
- reading and writing character data 1315
- stream to read text lines 1346

directory hierarchy

- breadth-first traversal 1348
- cyclic path dependency 1352
- depth 1347
- depth-first traversal 1348
- finding/searching 1354
- limit traversal by depth 1351
- root 1347
- siblings 1347
- traversal strategies 1348
- walking/traversing 1350

directory-level separator 1189

`DirectoryNotEmptyException` 1307

disjunct collections 808

dispensing canned advice application 1179

distance of shift 86

distributed 4

divide-and-conquer

- base case 1448
- forking 1448
- joining 1448
- parallel processing 1448

divide-and-conquer algorithm 868

divide-and-conquer paradigm 1447

dividend 62

division

- floating-point 62
- integer 62

division operator

- `/` 62

divisor 62

DML 1514

documentation 40

documentation comment 40

- tags 40

documenting *see* documentation

Domain Name System (DNS) 327, 1171

`DosFileAttributes` interface 1330, 1333

`DosFileAttributeView` interface 1334, 1338

dot 326

`double` 36, 42

double quote 38

`DoubleBinaryOperator` interface 722

`DoubleConsumer` functional interface 709

double-ended queues

  deque 821

`DoubleFunction` interface 712

`DoublePredicate` functional interface 703

`DoubleStream` interface 889

`DoubleSummaryStatistics` 890

`DoubleSupplier` functional interface 699

`DoubleToIntFunction` interface 713

`DoubleToLongFunction` interface 713

`DoubleUnaryOperator` interface 720

  composing unary operators 721

doubly linked list 788

`do-while` statement 173

downcasting 45

downstream collectors 979, 986, 1003

  default 985
  filtering adapter 992
  finishing adapter 997
  flat mapping adapter 994
  mapping adapter 993

DQL 1514

dropping stream elements 910, 913

duplicate elements 801, 804

duplicating objects 427

duration

  accessing units 1067
  comparable 1068

    - compared with period 1072
    - creating 1065
    - equality 1068
    - get methods 1067
    - immutable 1057, 1064
    - ISO format 1067
    - parsing 1066
    - plus/minus methods 1069
    - supported `ChronoUnit`s 1066, 1068, 1070
    - temporal arithmetic 1069
    - thread-safe 1057, 1064
    - time-based 1057, 1064
    - unit conversion 1068
    - `with` methods 1069
- `Duration` class 1064
    - *see* duration
- dynamic 4
- dynamic arrays 645
- dynamic binding
    - *see* dynamic method lookup
- dynamic method lookup 206, 278, 283
- dynamic type 196, 203
- dynamically resizable arrays 802

E

- eager execution 887
- effectively `final` 514, 687
- element type 118
- element type values for `@Target` 1569
- elements 117, 644, 783

elements of collections 783
eligible for garbage collection 534
ellipsis 136
    `else` clause matching 155
embedded applications 2
empty set 804
empty statement 101
empty string 441
encapsulation 3, 324, 326, 501, 1163, 1606
    nested types 493
encapsulation of implementation 195
enclosing class 492, 500, 502
enclosing context 493, 500, 505
enclosing instance 493, 498, 524
enclosing object reference 503
encoding of characters 1241
encounter order 891
end of file
    see end of I/O stream
end of file (EOF) 371
end of I/O stream 1234, 1240, 1243, 1252, 1253, 1261
ends with characters 452
enhanced `for` loop 172
enterprise applications 2
entries 785, 830
entry point 19, 22, 141, 1189, 1190, 1191, 1212
entry set 831, 1380, 1388
enum class
    see enum types
enum constants 287

  - symbolic names 287
  - values 287

enum types 163, 256, 287, 332, 635

  - abstract method declarations 294
  - accessibility 345
  - canonical form 288
  - constant-specific class bodies 294
  - declaring 288
  - enum constants 287
  - enum constructors 290
  - equality 293
  - final methods 293
  - implementing interfaces 298
  - implicit static methods 292
  - in `switch` expression 297
  - inherited methods 293
  - initializer blocks 291
  - local 494, 519
  - members 290
  - named constants 287
  - natural order 293
  - nested 492
  - ordinal value 163, 293
  - permitted direct subtypes 316
  - selector expression 156
  - static member 494, 495
  - top-level 491, 494
  - using 289

enumerated types

  - *see* enum types

`EOFException` 371

epoch 1024, 1049
epoch-second 1049
equality 75, 426, 744
  `equals` method 77, 426, 744
  object value 77
  objects 77
  primitive values 75
  reference values 76
  `equals` method 77, 426, 744
equivalence relation 746
equivalent collections 808
erasure 613, 616, 619
  of the supertype method 621
  same 620
`Error` 374
escape character 1101
escape sequences 38, 459
essential whitespace 462
evaluation order 52, 80
  arithmetic expressions 59
evaluation short-circuits 80
event listeners
  anonymous classes 521
exact integer arithmetic functions 481
exam 1615
  multiple-choice 1621
  Oracle exam attempt 1617
  program 1618
  questions 1619

    registration 1616
    result 1619
exam objectives
    OCP Java SE 11 1629
    OCP Java SE 17 1623
exam question assumptions
    OCP Java SE 17 1619
`Exception` 370, 372
exception handler 365
    see also exceptions
exception handling
    advantages 416
exception type 388
exceptions 365, 376
    `ArrayStoreException` 798
    backtrace 405
    cause 405
    chaining 405
    checked 371
    `ClassNotFoundException` 1262
    `ConcurrentModificationException` 795
    customized 375
    default handler 367
    detail message 370
    `EOFException` 1238
    `FileNotFoundException` 1236
    handler 365
    `IllegalThreadStateException` 1378
    `IOException` 1234, 1243

  - propagation 365
  - rethrowing 401
  - situations 370
  - suppressed 415
  - `throw` 386
  - throwing *see* throwing exceptions
  - thrown by JVM 370
  - thrown by method 100
  - thrown programmatically 370
  - `throws` 388
  - types 368
  - uncaught 367
  - unchecked 374
  - `UnsupportedOperationException` 790
- exchanging information 127
- execution 1370
- execution paths 1367
- execution stack
  - *see* JVM stack
- `Executor` 1424
- executor 1423
- Executor Framework 1423
- executor service 1427
  - bulk task execution 1439
  - creating 1429, 1431
  - executor lifecycle 1427
  - shutdown 1429, 1431, 1433
  - submitting tasks 1429, 1431
  - task cancellation 1444
  - task execution policy 1426
  - tasks 1423

  - termination 1429, 1431, 1433
  - thread pool 1425

- `Executors` 1424, 1425
- `ExecutorService` 1427
- explicit garbage collection 537
- explicit module 1206, 1208
- explicit referencing 515
- exploded module 1181
- exploded module directory 1181
- exponent 36
- exponential functions 480
- `exports` directive 1169, 1171, 1178
  - accessibility 1173
- `exports-to` directive 1178
- expression statements 55, 70, 101, 160, 175, 176, 682
- expressions 166
  - actual parameters 128
  - `boolean` 74
  - conditional 90
  - deterministic evaluation 50
  - `return` 184
  - statements 101
  - `switch` expression 164
  - `throw` 386
  - `yield` 164
- extending 521
  - inner class 508
  - interfaces 244
  - `Thread` 1375
- `extends` 581, 589, 591

- bounded type parameters 591
- constraint 591

`extends` clause 13, 191

extensibility 1607

extensions

- `.class` 21
- `.java` 21, 22

external iteration 889

extracting substrings 453, 466

F

fairness 1408

fairness policy 1461, 1470

- acquisition order 1461

fallback 1110

fall-through 156, 157, 165

- `false` literal 32, 37

fetch size 1536

FF *see* form feed

field declarations 99

field hiding 203, 231

field initialization 555

fields 6

- `final` 229
- hiding from superclass 203
- shadowing by local variables 114

FIFO queue 821

file 761

  see also directory entry

file attribute views 1329, 1333, 1334

  view name 1328

file attributes 1321

  accessing by attribute name 1327
  accessing by bulk operations 1328
  accessing individually 1321
  file attribute name format 1328
  file attribute views 1329
  file attributes interfaces 1330

file attributes interfaces 1328, 1330
`File` class 1290, 1292
file extension 1288

  `.properties` 1103

file name 325
file path 335

  separator character 335

file streams 1236
file system 1233, 1287

  accessing 1610
  default 1289

file system resources 1345
file visit option 1349
`FileAlreadyExistsException` 1309, 1339
`FileAttribute` interface 1339
`FileAttributeView` interface 1334
`FileInputStream` class 1235
`FileNotFoundException` 371

`FileOutputStream` class 1236
`FileOwnerAttributeView` interface 1334
`FileReader` class 1243
files
    binary 1238
    low-level I/O streams 1236
    not found 1236
`Files` class 1287
    method characteristics 1304
    streams 903
`FileSystem` class 1289
`FileSystems` class 1289
`FileTime` class 1324
`FileVisitOption` enum type 1349
`FileWriter` class 1244
filtering 688
filtering adapter 992
filtering streams 910
    using a `Predicate` 912
`FilterInputStream` class 1235
`FilterOutputStream` class 1236
filters 910
    see also I/O filter streams
final
    effectively 525, 692
    variables 525, 692
`final` fields 229, 230
    hiding 231

`final` instance fields initialization 231
`final` keyword 225
`final` local variables 229, 231
`final` members 225
`final` methods 226
    binding of method call 227
`final` parameters 135, 232
`final static` fields
    initialization 231
    manifest constant 231
`final` variables 229
    blank 229
    constant variables 229
    definite assignment analysis 232
    definitely assigned 232
    definitely unassigned 233
    fields 229
    local variables 229
    object state 229
`finalize` method 427
`finally` clause 377, 382
finishing adapter 997
finite streams 892
    extracting lines from a `String` 902
    from a `BufferedReader` 902
    from a `CharSequence` 901
    from arrays 898
    from collections 897
    from factory methods in the `Files` class 903
    from specified values 893

  from the `Random` class 900

first element 810, 811, 1486
first map entry 846
fixed arity method 136
fixed arity method call 140
Fixed Thread Pool 1426
fixed-size list view 659
flat mapping adapter 994
`float` 36, 42
floating-point 41

  `double` 42
  `float` 42
  literals 36

floating-point arithmetic 60
floating-point data types 36
floating-point division 62
floating-point remainder 63
floor entry 846
floor key 846
flow control *see* control flow
flushing I/O streams 1234
`for(;;)` statement 174

  backward 175
  collections 794
  forward 175
  iterating over array 121
  iterator 794

`for(:)` statement 176, 795

  iterating over array 121

for-each loop 172

Fork/Join Framework 1009, 1447
forking 1448
fork-join pool 1449
`ForkJoinPool` 1448
`ForkJoinTask` 1448
form feed 38, 39
formal parameters 100, 109, 128, 265, 354
  modifier 100
  name 100
  type 100
formal type parameter 567
`Format` class 1115
format elements 1140
  format style 1140
  format type 1140
format specifications 25, 457
format specifiers 25, 1247
format style 1127, 1132, 1140
format type 1140
`FormatStyle` enum type 1127, 1131
formatted output 24
  format specifiers 25
formatted prompting 1257
formatted string 457
formatted values
  `Console` 1257
  `PrintStream` 1247
  `PrintWriter` 1247
`Formatter` 1152
formatter

  - destination object 1152

- formatting 39, 1115
- formatting date and time values 1127
- formatting messages 1139
  - conditional formatting 1145
- formatting numbers/currency values 1116
- forward reference 541, 542, 545, 551, 555
- fractional signed numbers 41
- frame stack
  - *see* JVM stack
- framework for collections 783
- full permissions 1343
- fully 1191
- fully qualified class name 337, 1189, 1190, 1191
- fully qualified name 314, 345, 497
- fully qualified package name 326, 328, 329
- fully qualified type name 326, 331
- function 684
- function arrow `->` 679
- `Function` interface 712
- composing functions 715
- function type 683
- functional interface
  - `@FunctionalInterface` 675, 1579
  - abstract method 675
  - built-in functional interfaces 695
  - function type 683
  - functional method 675
  - primitive values 696
  - target type 682

    - *see also* lambda expressions

- functional method 675
- functional programming 5
- functional reduction 955, 956, 972, 984, 998
- functional requirements (FRs) 1600
- `FunctionalInterface` annotation 675
- functionality 675
- functional-style programming 675
- functional-style programming paradigm 3
- functions
    - currying functions 723

G

- garbage collection 536, 537
    - automatic 533
    - facilitate 537
- general abstractions 194
- general contract
    - `compareTo()` 762
    - `equals()` 746
    - `hashCode()` 756
- general loops 174
- generalization 13
- generalized classes 194
- generic class 567, 568
- generic instance method 594
- generic interface 572
- generic method 593, 610, 657
    - calling 594

generic method call

  inference 594

generic static method 594
generic subtype method 621
generic type 567, 645

  extending 573
  implementing 598

generics 147

  arrays 627
  casting 625
  class literals 636
  collections 578
  constructors 593
  enums 635
  erasure 613, 620
  exception handling 632
  inherited methods 620
  `instanceof` operator 624
  JVM 565
  method signature 615
  methods 593
  nested classes 633
  non-reifiable types 624
  overloading 615, 616
  overriding 615, 617
  reifiable types 623, 624
  subsignature 616
  type erasure 576, 613
  variable arity parameter 630
  wildcards 579

GESS 590

get methods 300, 302, 358, 590, 975, 1032, 1099, 1531, 1534, 1536, 1537

get operation 586

getter methods *see* get methods

GMT (Greenwich Mean Time) 1073

`goto` 32

grammar rules 30

grant 1608

graph

- nodes 1166
- unidirectional edges 1166

grouping 326

grouping separator 1125

groups of objects 783

GUI applications 1378

H

half-open interval 865, 867, 1145

handle-or-declare rule 388

handles *see* references

happens-before relationship 1414, 1454

- transitivity 1414

hard link

- *see* symbolic link

has-a relationship 194

hash code 108, 426, 753, 754

hash function 753

hash sign 1101

hash table 754, 788, 789

  - *see* map

hash value 426

  - *see* hash code

hashing 753

  - collision 753
  - hash code 753

`HashSet` 806

Haskell Curry 724

head map view 845, 846

head set view 810, 811, 1486

heap 533, 1370

heap pollution 1585

heap pollution warning 630, 650, 659, 805, 833, 863

helper methods 252

heuristics 756

hidden members

  - accessing 505, 506

hiding internals 324

hierarchical file system 1233

higher entry 846

higher key 846

high-level I/O streams 1233

high-performance 5

holdability 1540

horizontal tab 38

hotspots 5

HT *see* horizontal tab

human time 1049

I

I/O buffering 1238
I/O filter streams 1235, 1236, 1238
I/O package 1233
I/O streams 1233

- buffering 1238
- chaining 1238
- closing 1234, 1239, 1240
- end of stream 1240
- files 1236
- filter streams 1238
- flushing 1234
- high-level 1233
- input filter streams 1235
- low-level 1233
- of binary values 1236, 1238
- of characters 1241
- of objects 1236
- output filter streams 1236
- skipping bytes 1238

IANA Time Zone Database 1073
IDE (integrated development environment) 1616
idempotent method 408, 413
identifiers 30

- variable 102

identity container 965
identity conversion 46, 67
identity of object 8
`identity()` function 982
IEEE 754-1985 42
`if` block 152
`if-else` statement 153

  - scope of pattern variable 276

- illegal thread state 1378
- `IllegalArgumentException` 372, 810, 1134
- `IllegalStateException` 373, 794
- immediately enclosing instance 492
- immutability 1453, 1607
- immutable 302, 796, 1024, 1075
- immutable classes
  - in Java SE platform documentation 356
- immutable object pattern
  - *see* immutable objects
- immutable objects 429, 440
  - advantages 360
  - caching 358
  - consistency of state 358
  - defensive copying 359
  - guidelines 358
  - state 356
  - static factory method 359
  - thread-safety 356
- immutable primitive values 356
- immutable strings 440
- implementation inheritance 245
- implementations 6, 193, 324
  - collections 783
  - inheritance hierarchy 226
- implementations of collections 785
- implementing
  - `Comparable` 761

    `Comparator` 769
    equals 744
    hashCode 753
    interfaces 240

`implements` clause 240
implicit canonical record constructor 302
implicit default constructor 110
implicit dependency 1167, 1172, 1216
implicit inheritance 191
implicit narrowing conversions 67
implied dependency 1175
`import` 345

    declaration 329
    see also static import
    single-type-import declaration 329
    statement 325
    type-import-on-demand declaration 330

importing

    enum constants 332
    nested types 498
    reference types 329
    static members 331

in lockstep 797
IN parameters 1530
incidental whitespace 462
increment operator 69
indenting lines 456
index 117
index expression 120
`IndexOutOfBoundsException` 400, 444, 445, 454, 465, 467, 468, 651, 652, 655

individual array elements 120
individual module compilation 1186
inequality 75

  see also equality

inferred-type lambda parameters 147
inferred-type parameters 680
infinite loop 176
infinite streams 892

  building with generator functions 894
  from the `Random` class 900

infinity 60, 432

  negative 60
  positive 60

information hiding 324
inheritance 13, 194, 238

  hierarchy 193, 507
  serialization 1270
  subtype–supertype relationship 195

inheritance of implementation 238
inheritance of type 238
initial capacity 647
initial state of object 555
initialization

  arrays 119, 124
  code 120
  default values 103
  `for` statement 174
  objects 8
  references 102

  - variables 102

- initialization expression 102, 142
- initializer 540
  - declaration-before-reading rule 542
  - static 541, 542, 553, 554, 557
- initializer blocks
  - instance 551
  - static 545
- initializer expression 103, 229, 230, 540
  - static 543
- initializers
  - declaration order 545
  - exception handling 549
  - forward references 545
  - non-static block 99
  - non-static field 99
  - static block 99
  - static field 99
  - using class name 547
  - using simple name 545
- initializing *see* initialization
- inlining 227
- inner classes 491, 492
  - anonymous 492, 494, 521
  - local 492, 494, 512
  - non-static member 492, 494
  - qualified superclass constructor invocation 509
  - synchronization 1392

INOUT parameters 1532

input 1233

input stream 1233

input validation 1605

`InputStream` abstract class 1234

`InputStreamReader` class 1243

`INSERT` statement 1515

insertion order 644, 789, 807, 842

insertion point 861, 867

instance

- members 13
- methods 13, 101, 106
- variable initialization 103
- variables 13, 106
- *see also* object

instance initializer blocks 551

- local variable type inference 552

instance methods 9

instance variables 8, 555

`instanceof` operators 92, 269, 274, 624

`instanceof` pattern match operator 274

- pattern variable 275
- type pattern 275

`instanceof` type comparison operator 270, 281

instant

- accessing fields 1052
- comparing 1053
- convert to temporal objects 1056
- creating 1049
- creating from date-time 1051

    current instant 1050
    epoch 1049
    epoch-second 1049
    immutable 1049
    ISO format 1050
    max value 1049
    min value 1049
    nano-of-second 1049
    parsing 1051
    plus/minus methods 1054
    representation 1049
    supported `ChronoField`s 1054
    supported `ChronoUnit`s 1056
    temporal arithmetic 1054
    text representation 1051
    thread-safe 1049
    timestamps 1049
    `with` methods 1054
    zero UTC offset 1049

`Instant` class

    see instant

instant timeline 1074
instantiation 8

    anonymous classes 524
    local classes 517
    non-static member classes 502
    parameterized type 569
    static member classes 495

`int` 32, 42
`IntBinaryOperator` interface 722
`IntConsumer` functional interface 709

integer arithmetic 59
integer bitwise operators 82
integer constant expressions 48
integer data types 32
integer division 62
integer remainder operation 62
integers 41

- *and* operator 83
- `byte` 42
- complement 83
- data types 41
- `int` 42
- literals 32
- `long` 42
- *or* operator 83
- representation 54
- `short` 42
- types 41
- *xor* 83

integral types 41, 42, 44
interface constant antipattern 332
interface-based 787
interfaces

- abstract 239
- abstract methods 240
- accessibility 345
- body 238
- collection core 783
- constants 254
- contract 240
- declarations 325

  - default methods 246
  - evolution 250
  - extending 244
  - functional interfaces 675
  - header 238
  - implementing 240, 298, 523
  - inheriting from multiple interfaces 621
  - initialization 557
  - local 494, 519
  - markers 239
  - member declarations 238, 239
  - multiple inheritance of implementation 248
  - nested 491, 492
  - overriding default methods 247
  - partial implementation 242
  - private methods 252
  - public methods from `Object` class 245
  - realization 244
  - references 246
  - static member 494, 495
  - static methods 251
  - subinterfaces 244
  - superinterfaces 244
  - top-level 491, 494
  - UML 244
  - UML stereotype 244
  - variables 254
  - with single abstract method 678

- intermediate stream operations 885, 905
  - examining elements 920
  - filtering streams 912
  - flattening streams 924

    - input stream 885
    - lazy execution 887
    - mapping streams 921
    - non-interfering behavioral parameters 910
    - order of 908
    - output stream 885
    - replacing each element with multiple elements 927
    - selecting distinct elements 915
    - setting execution mode 933
    - setting stream as unordered 932
    - short-circuit execution 908, 917, 918
    - skipping elements 915
    - sorting streams 929
    - stateful operations 908
    - stateless behavioral parameters 910
    - stateless operations 908
    - truncating a stream 917
- internal iteration 889
- internal packages 1171
- internationalization 1096
- interned strings 440, 442
- interned values 433
- interpackage accessibility 324
- interrupt handling 1393
    - interrupt status 1393
    - `InterruptedException` 1393
    - spurious wakeup 1399
- `InterruptedException` 1393
- interruptible locking 1462, 1467
- intersection 804
- `IntFunction` interface 712

`IntPredicate` functional interface 703

intrinsic lock

  see object lock

intrinsic locking 1388, 1460

`IntStream` interface 889

`IntSummaryStatistics` 890

`IntSupplier` functional interface 699

`IntToDoubleFunction` interface 713

`IntToLongFunction` interface 712

`IntUnaryOperator` interface 720

  composing unary operators 721

invocation

  parameterized type 569

invocation stack

  see JVM stack

invoker 184

invoking garbage collection 537

`IOException` 371

is-a relationship 194, 195

ISO 8859-1 encoding scheme 439

ISO Latin-1 37, 42

ISO standard 1127, 1129

`Iterable` interface 450, 657, 792, 795

iteration 174, 792

iteration order 795

iteration statements 172

  next iteration 182

  termination 172, 181

iterators 645, 657, 793, 797

  - fail-fast 794, 1485
  - `ListIterator` 801
  - ordered collection 796
  - snapshot-style 1484
  - sorted collection 796
  - unordered collection 796
  - weakly consistent 1484

J

- Jakarta EE (Enterprise Edition) 2
- JAR 1163, 1189
- `jar` tool 1189
- `java` 21
- Java Application Launcher 21
  - *see* JDK tool `java`
- Java Archive
  - *see* JAR
- Java Archive Tool
  - *see* JDK tool `jar`
- Java bytecode 21
- Java Card 2
- Java Class Dependency Analyzer Tool
  - *see* JDK tool `jdeps`
- Java Collections Framework 644
- Java compilation unit 1168
- Java Database Connectivity
  - *see* JDBC
- Java Development Kit (JDK) 2

Java ecosystem 2

Java EE (Enterprise Edition) 2

Java Language Compiler 20

see JDK tool `javac`

Java Linker

see JDK tool `jlink`

Java ME (Micro Edition) 2

Java Platform Module System 1163

Java Platforms 2

Java Runtime Environment (JRE) 3

Java SE (Standard Edition) 2

Java Shell Tool (`jshell`) 23

Java Virtual Machine (JVM) 2

`java.base` module 1167, 1214

`java.io` package 1233, 1290

`java.lang.annotation` package 1557

`java.lang.Record` class 304

`java.lang.reflect` package 1587

`java.math` package 485

`java.net` package 1290

`java.nio` package 1287

`java.nio.file` package 1287

`java.nio.file.attribute` package 1324

`java.se` module 1168

`java.sql` package 1517

`java.text` package 1115

`java.time` package 1024, 1115

`java.time.format` package 1115

`java.util` package 644

`java.util.concurrent` package 1424

`java.util.function` package 695

`javac` 20

Javadoc comment 40

- `@param` tag 100, 112
- `@return` tag 184
- `@throws` tag 390

`javadoc` tool 40

JDBC 1517

JDBC API 1517

JDBC drivers 1517

- legacy drivers 1522

JDBC interfaces 1519

JDBC resources 1518

JDBC URL 1519

JDK 1616

- licencing 3

JDK tools

- `jar` 1189, 1211, 1212, 1213, 1221
- `java` 21, 1189, 1212, 1213, 1220
- `javac` 20, 1186, 1219
- `javadoc` 40
- `jdeps` 1167, 1214, 1222
- `jlink` 1204, 1222
- `jmod` 1166
- `jshell` 23
- summary of operations 1218

JMOD archive 1166

JMOD files 1213

join

- starvation 1412

joining 1448
joining threads 1383
JPMS 1163
just-in-time (JIT) 4
JVM 534, 537
  heap 1370
  JVM stack 1370
  method area 1370
  program counter (PC) 1370
  runtime constant pool 1370
  runtime data areas 1369
JVM stack 365, 534, 1370

K

key 866
key insertion order 841
key objects 785
key sets 831, 847
key sorting 845
key-based lookup 1104
keys 830
keystore 1608
key–value entries 754
key–value pairs 1102
keywords 31
  `abstract` 219, 240, 675
  `boolean` 43
  `break` statement 180
  `byte` 42
  `case` 156, 160, 164
  `catch` 377

`char` 42

`class` 99, 238

contextual words 31

`continue` 182

`default` 156, 161, 166, 246

`do` 173

`double` 42

`else` 153

`extends` 191, 581, 589, 591

`final` 135, 225, 232, 312, 358, 400, 410, 1456

`finally` 382

`float` 42

`for` 174, 176

`if` 152

`implements` 240

`import` 329, 345

`instanceof` 92, 270, 274, 281, 624

`int` 42

`interface` 238

`long` 42

`new` *see* `new` operator

`null` 49, 77, 269

`package` 328, 345

`private` 351

`protected` 350

`public` 350

reserved words 31

`return` 184

`short` 42

`static` 19, 113, 251, 331, 495, 545

`super` 110, 201, 206, 211, 248, 507, 582

`switch` 155, 164

`synchronized` 1388, 1456

- `this` 106, 504
- `throw` 386
- `throws` 388
- `try` 377
- unused words 31
- `void` 19, 429
- `volatile` 1454
- `while` 172, 173

keywords (contextual)

- `exports` 31, 1169, 1177
- `module` 31, 1177
- `non-sealed` 31
- `open` 31, 1177
- `opens` 31, 1177
- `permits` 31
- `provides` 31, 1177
- `record` 31
- `requires` 31, 1170, 1175, 1177
- `sealed` 31
- `to` 31, 1177
- `transitive` 31, 1177
- `uses` 31, 1177
- `var` 31
- `with` 31, 1177
- `yield` 31

keywords (reserved)

- `const` 32
- `goto` 32

L

labeled `break` statement 181

labels 179, 181

- `break` 181
- `case` constants 156, 160
- constant expressions 157
- `default` 156, 161, 166
- labeled statement 179
- `switch` statement 156, 160

lambda body 679, 681

lambda expressions 679, 691

- access class members 685
- block scope 684, 686
- compared with anonymous classes 692
- contexts providing target types 733
- declared-type parameters 680
- deferred execution 679, 684
- expression 681
- expression statement 682
- function 684
- function arrow `->` 679
- implicitly typed 680
- inferred-type parameters 680
- lambda body 679, 681
- lambda parameters 680
- non-`void` return 681
- parameter list 679
- shadowing members 685
- single expression 681
- statement block 682
- target type 682
- target typing 684
- type checking 682

  - using `this` and `super` 685
  - variable capture 687
  - var-type inferred parameters 680
  - `void` return 681

lambda parameters 680
language codes 1097
language tag 1118
last element 810, 811, 1486
last map entry 846
latch

  - *see* count-down latch

late binding

  - *see* dynamic method lookup

LATIN-1 encoding scheme 439, 440
lazy execution 887, 907
least significant bit 34
left associativity 51
left shift 86
legacy class 566
legacy code 565
legacy supertypes 619
legal assignments 264
`length` method 445, 902
letter pattern 1134
lexical nesting 496
lexical scope

  - *see* block scope

lexical tokens 30
lexicographic array comparison 869
lexicographic comparison 869

lexicographical ordering 447
LF *see* linefeed
lifetime 535
    see scope 106
limits 1146
line continuation 38
line separator 25, 1246
line terminator 39, 445
linear implementation inheritance 193
linear inheritance 237
line-continuation character 1186
linefeed 38
lines of text 1252
`LinkageError` 374
linked data structures 566
linked list 788, 802
`LinkedBlockingDeque` 1496
`LinkedBlockingQueue` 1496
`LinkedHashSet` 806
`LinkedList` 647, 802, 815, 823
`LinkedTransferQueue` 1496
`LinkOption` enum type 1301
`List` 645, 801
list views 656, 659, 660
listing observable modules 1212
`ListIterator` 801
lists 784, 801, 802
    `ArrayList` 802
    binary search 861
    changing contents 863
    copying 863

  - `LinkedList` 802
  - `List` 801
  - `ListIterator` 801
  - positional 785
  - replacing elements 862, 863
  - reverse order 859
  - sorted 861
  - sorting 858, 859
  - sublist view 656
  - sublists 862
  - unmodifiable 649, 801
  - unmodifiable views 856
  - `Vector` 802
  - views 659
  - *see also* `ArrayList`

- literals 32
  - boolean 37
  - character 37
  - default type 32, 36
  - double 36
  - escape sequences 38
  - `false` 37
  - float 36
  - floating-point 36
  - integer 32
  - `null` 32
  - predefined 31
  - prefix 33
  - quoting 37
  - scientific notation 36
  - string 39
  - suffix 32, 36

  - `true` 37

live threads 1383

livelock 1410

liveness 1408

local 102, 104, 519
  - block 492
  - chaining of constructors 210, 555
  - classes 494, 512
  - enum types 492, 494
  - interfaces 492, 494
  - record classes 492, 494
  - variables 106, 354

local classes 492, 494, 512
  - accessing local declarations 513
  - accessing members in enclosing context 515
  - declaration 512
  - hiding members from enclosing context 515
  - in non-static context 512
  - in static context 512
  - instantiation 517
  - scope 512
  - shadowing local variables 514
  - using qualified `this` 515

local date
  - conversion to `Date` 1088, 1141

local date-time
  - conversion to `Date` 1088, 1141
  - converting to instant 1051

local declarations 101

local enum types 519
local interfaces 519
local record class 519
local scope 169
local time 1073
  conversion to `Date` 1088, 1141
local timeline 1074
local variable type inference 142, 410
  arrays 143
  instance initializer blocks 552
  parameterized types 571
  static initializer blocks 545, 547
local variables 109, 142
  block scope 684
`LocalDate`
  see temporal objects
`LocalDateTime` class
  see temporal objects
`Locale` 1096
locale 448, 1096
  country codes 1097
  country name 1098
  default 1098
  language codes 1097
  language name 1098
  predefined 1097
locale qualifiers 1098
locale-sensitive 1115

locale-sensitive operations 1096
localization 1096
localizing an application 1105
localizing information 324
`LocalTime` class
    see temporal objects
locations
    see class path
`Lock` 1461
lock 1380
    acquire 1380, 1388
    class 1391
    entry set 1380, 1388
    release 1380
lock acquisition
    starvation 1412
logarithm 480
logical 277
logical AND
    `&` 78
logical complement
    `!` 78
logical complement (`!`) operator
    pattern variable 277
logical exclusive OR
    `^` 78
logical inclusive OR

  | 78

logical XOR

  ^ 78

`long` 32, 42

  suffix 32

`LongBinaryOperator` interface 722
`LongConsumer` functional interface 709
`LongFunction` interface 712
`LongPredicate` functional interface 703
`LongStream` interface 889
`LongSummaryStatistics` 890
`LongSupplier` functional interface 699
`LongToDoubleFunction` interface 713
`LongToIntFunction` interface 713
`LongUnaryOperator` interface 720

  composing unary operators 721

loop body 172, 174
loop condition 172, 174
loop variable 174
loops *see* iteration statements
loss of precision 44
lower bound 581
lower bounded wildcard

  `? super T` 582

lower bounded wildcard references 589
lower entry 846
lower key 846
low-level I/O streams 1233
LTS (long-term support) release 3

M

magnitude 44

main memory 1453

`main` method 21, 141

- arguments 141
- modifiers 141

main thread 1373, 1378

`Main-Class` attribute 1212

`MANIFEST.MF` file 1212

many-to-one relation 830

`Map` 831

map

- key–value entries 754

map views 839

`Map.Entry` 833, 846

mapping adapter 993

mappings 785

- *see entries* 785

map-reduce operation 1003

map-reduce transformation 958

maps 785, 830

- access order 841
- entries 785
- iteration 839
- key insertion order 841
- key-based operations 831, 835
- `Map` 831, 840
- map views 839
- mappings 785

  - multimap 597
  - `NavigableMap` 845
  - ordering mode 841
  - replacing values 839
  - `SortedMap` 845
  - submap view 846
  - text representation 841
  - `TreeMap` 847
  - unmodifiable 832
  - unmodifiable views 857
  - views 839

marker annotation 1563, 1565
marker annotation type 1560
marker interfaces 239, 1262
marker parameter 1526, 1530
mask

  - *see* bit mask

`Math` class 108, 478
math constants 478
mathematical set 804
`MAX_VALUE` constant 434
maximum value 479
member declarations 99, 307
member type declarations 99

  - non-static member class 492
  - static member class 492
  - static member enum type 492
  - static member interface 492
  - static record class 492

members 6, 352

  - access 106
  - accessibility 347
  - default values 103
  - inheritance 191
  - modified 191
  - of objects 9
  - scope 352
  - shorthand 107
  - `static` 10
  - terminology 13
  - variables *see* fields

- memory consistency errors 1414
- memory management 533
- memory organization 533
- `MessageFormat` 1115
- meta-annotation 1559, 1567
  - `@Documented` 1574
  - `@Inherited` 1574
  - `@Repeatable` 1575
  - `@Retention` 1567
  - `@Target` 1569
- metadata 239, 1557
- `META-INF` directory 1211, 1212
- method area 1370
- method call 9, 100, 127
  - chaining 467, 469
  - fixed arity 140
  - variable arity 139
- method chaining 906, 1036, 1042, 1060, 1069
- method declaration 99

  - variable arity parameter 630
- method header 224
- method invocation conversions 48, 265
- method modifiers 100
- method naming conventions 1025
- method overloading 108, 202, 225
- method overriding 202, 224, 226, 556
  - covariant return 201
- method references 724, 725
  - bounded instance method references 727
  - static method references 726
  - unbounded instance method references 729
- method signature 101, 196
- method type 683
- methods 6
  - `@Override` 198
  - `abstract` 224, 240, 675
  - abstract method declarations 240
  - advisory 1385, 1387
  - ambiguous call 202, 266
  - automatic variables *see* local variables
  - behavior 675
  - binding of method call 205
  - blocks 101
  - body 100
  - bridge 615
  - call chaining 128
  - call *see* method call
  - calling variable arity method 137
  - chained 449

`clone` 426
concrete 226
declaration 100, 128
`default` 246
dynamic lookup 283
`equals` 77, 426, 744
exceptions 100
`final` 226
`finalize` 427
fixed arity 136
functional 675
generic 593
`getClass` 426
header 100
hiding 226
hiding static methods 203
implementation 224
invocation *see* method call
local declarations 101
local variables
locale-sensitive 1099
`main` *see* main method
method invocation conversions 265
method type 683
modifiers 100
most specific 266, 655
mutually exclusive 1391
name 127
objects 106
overloaded resolution 265
overloading *see* method overloading
overriding *see* method overriding
overriding vs. overloading 202

  - parameters 100
  - `return` 184
  - return value 100
  - signature 101, 108, 202, 616
  - `static` 115, 251
  - subsignature 617
  - `synchronized` 1388
  - termination 184
  - `throws` clause 388
  - `toString` 426
  - variable arity 136
- migration to modules 1209
- `MIN_VALUE` constant 434
- minimizing overhead 536
- minimum value 478
- `MissingResourceException` 374, 1104
- mixing raw types and generic types 575
- mobile applications 2
- modifiers
  - `abstract` 240
  - access 345
  - `default` 246
  - `final` 225
  - `static` 251
- modular JAR 1189, 1206
  - creating 1189
  - describe a JAR 1213
  - extract contents 1212
  - list contents 1211
  - listing observable modules 1212
  - `Main-Class` attribute 1212

  `MANIFEST.MF` file 1212
  `META-INF` directory 1211, 1212
  running from 1191

modular JDK 3, 1164

  goals 1165
  `java.base` module 1167, 1214
  `java.se` module 1168

modular JDK installation 1165
module application directory structure 1180
module declaration 1168

  `module` as contextual keyword 1169

module dependency 1170
module descriptor 1169, 1187
module directives 1169

  `exports` directive 1169, 1171, 1178
  `exports-to` directive 1178
  `opens` directive 1178, 1191, 1192
  `opens-to` directive 1178, 1195
  overview of directives 1177
  `provides-with` directive 1178, 1200
  qualified export 1178
  `requires` directive 1169, 1170, 1177
  `requires static` directive 1177
  `requires transitive` directive 1175, 1177
  unqualified export 1169, 1178
  `uses` directive 1178, 1202

module graph 1166, 1172, 1180, 1217

  acyclic 1172
  dependency cycles 1172
  implicit dependency 1167, 1172

  - implied dependency 1175
  - module dependency 1170
  - sink nodes 1172

- module name 1171
- module path 1206
- module resolution 1172, 1186
- module root directory 1181
- `module-info.java` file 1181
- modules 1168

  - accessibility 1173
  - accessibility rules 1170, 1174
  - aggregate module 1168
  - automatic 1206, 1207
  - benefits 1163
  - comparison 1208
  - creating modular JAR 1189
  - dependencies 1166
  - explicit 1206, 1208
  - exploded module 1181
  - exploded module directory 1181
  - exploring modules 1211
  - implicit dependency 1216
  - individual module compilation 1186
  - internal packages 1171
  - migration to 1209
  - modular JAR 1189
  - module application directory structure 1180
  - module declaration 1168
  - module descriptor 1169
  - module graph 1166
  - module name 1171
  - module resolution 1186

  - module root directory 1181
  - `module-info.java` file 1181
  - multi-module compilation 1187
  - named 1208
  - non-standard modules 1166
  - `open` modifier 1178
  - platform modules 1166
  - principle exported package 1171
  - public API 1163, 1170, 1174
  - readability 1173
  - readable 1170
  - reliable configuration 1163
  - running from a modular JAR 1191
  - services 1196
  - standard modules 1166
  - stronger encapsulation 1163
  - system modules 1212
  - unnamed 1206
  - viewing dependencies 1214

- monitor 1380
  - entry set 1380
  - lock 1380

- monitor lock
  - *see* object lock

- `Month` enum type 1028
- most specific method 266, 655
- multi-`catch` clause 397
  - `final` parameter 400

- multicore architectures 5
- multidimensional arrays 122, 124

multi-element annotation type 1561
multilevel classification 987
multilevel grouping 987
multilevel partitioning 991
multimap 597
multi-module compilation 1187
multi-paradigm programming 3
multiple assignments 55
multiple bounds 592
multiple `catch` clauses 376, 397
multiple inheritance of implementation 193, 238, 248
multiple inheritance of type 238
multiple-line comment 40
multiplication operator `*` 61
multiplicative operators 61
multisets 804
multitasking 1367
multithreaded 5
multithreaded programming 1368
mutability 661
mutable character sequences 464
mutable containers 947, 964
mutable objects 359
mutable reduction 971, 972

    mutable containers 964
    see also collectors

mutator method *see* set methods
mutex

    see lock

mutual exclusion 1455
mutually comparable 858, 861, 867

mutually exclusive

  - actions 154
  - locks 1380

MVC design pattern 1179

N

name 30
name clash 620, 621, 622
name elements 1288
name separator in path 1288, 1291
named constants 231
named modules 314, 1208
named package 314
namespaces 109, 1171
naming conventions of paths 1288
NaN (Not-a-Number) 61, 432, 1606
nano-of-second 1049
narrower range 44
narrowing conversions

  - primitive 44
  - reference 45

narrowing reference conversions 196, 270, 585, 605
natural logarithms 478, 480
natural order 761, 858, 865
natural ordering 865
navigable map 785

  - first entry 846
  - last entry 846
  - range-views 846
  - views 846

navigable set 784, 810

- ceiling 812, 1487
- floor 812, 1487
- higher 812, 1487
- iterator 812, 1487
- lower 812, 1487

`NavigableMap` 845
`NavigableSet` 810
navigation methods 845

- closest match 811

nbsp

- *see* non-breaking space

negative zero 61
nested classes 492

- anonymous 492, 494, 521
- class file name 497
- fully qualified name 497
- generic 633
- local 492, 494, 512
- non-static member 492, 494, 501
- qualified name 496
- static member 491, 494, 495
- synchronization 1392

nested enum types

- local 494
- static member 491, 494

nested interfaces 491

- static member 494

nested lists 648

nested loops 126

nested parameterization 575

nested parameterized types 608

nested record classes 492

  - local 494
  - static member 494

nested type declaration 491

nested types 491

  - accessibility 493
  - enclosing context 493
  - enclosing instance 493
  - importing 498

nested wildcards 584

`new` operator 8, 92, 109, 119, 507, 521, 555

  - qualified class instance creation expression 502

`NEW` state 1382, 1386

newline *see* linefeed

NIO.2 1287

NL *see* newline

no-argument constructor 110, 210, 213

`NoClassDefFoundError` 374

nodes 566, 1166

non-access method modifier 99, 100

non-associativity 50

non-atomic operation 1455

non-blocking collection 1483

non-blocking lock-free synchronization 1457

non-breaking space (nbsp) 1117, 1120

non-functional requirements (NFRs) 1600

non-interfering behavioral parameters 910
non-modular JARS

    see plain JARs

non-observable substate 1382
non-reifiable types 623, 631
non-runnable states 1383
non-standard modules 1166
non-static code 100

    see non-static context 99

non-static context 99, 100
non-static field 13
non-static field initializers 99
non-static initializer block 99
non-static member classes 494

    access hidden members 505
    access in enclosing context 504
    declaration 501
    fully qualified name 497
    hiding in enclosing context 507
    inheritance 507
    instantiation 502
    qualified class instance creation expression 502, 503
    qualified name 496
    qualified superclass constructor invocation 509
    qualified `this` 504

non-static nested classes

    see inner classes

non-static type 568
non-varargs call

  see fixed arity call

non-`void` return 681

non-zero argument constructor 111, 214

normal annotation 1563
  array element initializer 1564
  element-value pair 1563
  single-element array-valued element-value pair 1564

normal canonical record constructor 304, 305

normal execution 366

normal non-canonical record constructor 305, 307

normal threads 1377

`NoSuchElementException` 793, 811, 815

`NoSuchFileException` 1307

Not-a-Number
  see NAN

notifying threads 427, 1396

`NotSerializableException` 371

`null` literal 32
  casting 49
  equality comparison 77

null reference 32
  casting 269

nulling references 536

`NullPointerException` 373

`null`-safe 743

`Number` class 434

number format pattern symbols 1125

number formatter 1117, 1119

  - accounting currency 1118
  - compact form 1120
  - currency 1116, 1117
  - numbers 1116
  - percentage 1116, 1117

number formatting

  - accounting currency 1118
  - compact form 1120
  - compact form styles 1120
  - pattern symbols 1125
  - static factory methods 1116
  - string pattern 1125

number systems

  - base 33
  - decimal 33
  - hexadecimal 33
  - octal 33
  - radix 33

`NumberFormat` 1115, 1116
`NumberFormatException` 373, 397, 431
numbers 1096, 1116

  - formatting 1116

numeric optional classes 944
numeric promotions 49

  - assignment 56
  - binary 50
  - unary 49

numeric streams 885, 892

  - averaging 974

- from a range 898
- from arrays 898
- from the `Random` class 900
- functional reduction 972
- summary statistics 972, 974
- summation 973

numeric types 41

numeric wrapper classes 434

numerical literals

- using underscore 36

O

`ObjDoubleConsumer` functional interface 709

`Object` class 193, 425

object creation 1270

object graph 1266

object hierarchy 195

object lock 1380, 1388, 1391, 1397, 1398, 1399, 1408

Object lock rule 1414

object reference equality 745

object references 8, 102

object serialization

- *see* serialization

object state 8, 109, 131, 555

- immutable 356, 1453

object streams 885, 892

object type *see* dynamic type

object value equality 745

`ObjectInputStream` class 1235

object-level lock

  see object lock

object-oriented programming 6

object-oriented programming paradigm 3

`ObjectOutputStream` class 1236, 1261

objects 8

  aggregate 16

  alive 534

  arrays 118

  behaviors 6

  callee 127

  caller 127

  `Class` class 426

  cleaning up 536

  cloning 427

  communication 127

  comparing 426, 744, 769

  compound 534

  constituent 16, 534

  constructing 555

  contract 324

  creation 1270

  decoupling 283

  deserialization 1261

  eligible 536

  equality 77, 426, 744

  exchanging information 127

  garbage collection 533

  I/O streams 1236

  identity 8

  immutable 429

  implementation 324

  initial state 555

- initialization 8, 109
- initializer block 551
- internals 324
- lifetime 535
- members 9
- methods 106
- `Object` class 425
- persistence 1261
- properties 6
- reachable 533, 534
- reading 1261
- serialization 1261
- services 324
- state 229
- state *see* object state
- value equality 77
- wait set for threads 1396
- writing 1261

`Objects` class

- comparing objects 743
- computing hash code 743
- `null`-safe methods 743

`ObjIntConsumer` functional interface 709

`ObjLongConsumer` functional interface 709

observable modules 1213

on/off latch 1475

one’s complement 34

one-arity 695

one-dimensional arrays 118

one-shot tasks 1440

one-to-many mapping 924, 927

one-to-one mapping 921
one-to-one transformation 927
online proctored exam 1618
on-the-fly classes 492
`open` modifier 1178, 1196
open module 1196
open package 1192
open range-view operations 801
opening delimiter 458
`OpenOption` interface 1301
`opens` directive 1178, 1191, 1192
`opens-to` directive 1178, 1195
operands 48

    evaluation order 52

operations 6

    put 586
    read 586
    write 586

operators 50

    `-` 58, 63
    `--` 69
    `^` 78, 82
    `^=` 79, 85
    `!` 78
    `!=` 75, 76
    `? :` 90
    `.` 9, 326
    `[]` 92, 120
    `*` 58, 61
    `*=` 66
    `/` 58, 61, 62

/= 66

& 78, 82

&& 80

&= 79, 85

% 58, 61, 62

%= 66

+ 58, 63, 67

++ 69

+= 66

< 74

<< 86, 87

<<= 90

<= 74

-= 66

= 54

== 75, 76

-> 92

> 74

>= 74

>> 86, 88

>>= 90

>>> 86, 89

>>>= 90

| 78, 82

|= 79, 85

|| 80

~ 82

arithmetic compound assignment 66

assignment 50, 54

associativity 50

binary 50

bitwise 82, 86

boolean 74, 75, 78

cast 50
comparisons 74
compound assignment 79, 85
conditional 80, 90
decrement 69
dot 9
equality 75
execution order 52
extended assignments 90
floating-point 60
floating-point division 62
floating-point remainder 63
increment 69
`instanceof` 92
`instanceof` pattern match operator 274
`instanceof` type comparison operator 270
integer 82
integer arithmetic 59
integer division 62
integer remainder 62
logical 78
multiplicative 61
`new` *see* `new` operator
overflow 60
overloaded 59, 62
postfix 50
precedence 50
relational 74
shift 86
short-circuited 80
string concatenation 67
ternary 50
unary 50, 61

  - unary `-` 61
  - unary `+` 61

- optimizations 5
- optional annotation element 1562, 1566
- `Optional` class 890, 940
  - absence of a value 940
  - creating 941
  - declaring and returning 940
  - querying 943
  - *see also* numeric optional classes 944
- optional dependency 1177
- optional operations 790
- `OptionalDouble` 890, 944
- `OptionalInt` 890, 944
- `OptionalLong` 890, 944
- optionally bounded collections 1483
- *or* operator 83
- Oracle CertView 1617
- Oracle exam attempt 1617
- Oracle JDK 3
- Oracle MyLearn 1617
- ordered collections 784, 801
- ordered streams 891
  - extracting lines from a `String` 902
  - from a `BufferedReader` 902
  - from a `CharSequence` 901
  - from factory methods in the `Files` class 903
- ordering mode 841
- ordinal value 163, 293
- OUT parameters 1530
- outer instance 508, 509

`OutOfMemoryError` 374, 538
output 24, 1233
output stream 1233
`OutputStream` abstract class 1234
`OutputStreamWriter` class 1244
overflow 34, 60
overflow errors 481
overloaded 59
overloaded method resolution 265
overloaded methods
    ambiguous call 202
overloading 616
    constructors 112
    method resolution 265
    methods 108, 202
overloading vs. overriding 202
override criteria 617
override-equivalent 616, 621
overriding 390, 621
    `@Override` 618, 1578
    covariant return
    `equals` 77, 744
    `hashCode` 753
    methods 202
    override-equivalent 616
    `toString` 68
overriding vs. overloading 202

P

package accessibility for class members 347, 351

package accessibility for top-level types 345
package directory 335
`package` statement 325, 328, 345
package-private accessibility for class members
    see package accessibility for class members
package-private accessibility for top-level types
    see package accessibility for top-level types
packages 326
    classpath 339
    declaration 325
    definition 328
    destination directory 335
    hierarchy 326
    `java.io` 1233
    `java.lang` 425
    `java.util` 783
    location 339
    members 326
    package directory 335
    root 339
    running code from 337
    shorthand 329
    statement *see* package statement
    unnamed 328
    using 329
palindromes 688
parallel processing 1448
parallel streams 891, 1009
    autoboxing 1015
    behavioral parameters 910

    - benchmarking 1010
    - building parallel streams 934, 1009
    - collecting to a `ConcurrentMap` 983
    - computation-intensive operations 1010
    - concatenating 896
    - concurrent collectors 983, 989
    - data size 1010
    - determine if parallel 934
    - encounter order 953
    - execution 962, 966, 1009
    - factors affecting performance 1010
    - from collections 897
    - grouping to a `ConcurrentMap` 989
    - ordered 892
    - ordering 1015
    - parallel functional reduction 962
    - parallel mutable reduction 966
    - setting parallel execution mode 933, 934
    - shared state 1014
    - side effects 910, 948
        - non-interfering behaviors 1014
        - stateless behaviors 1014
    - splittable 1010
    - stateful operations 911, 912, 1014, 1015
    - unordered 892
- parallelism 1368
- parameter list *see* formal parameters
- parameter passing
    - by value 128
    - variable arity 136
- parameterized types 569, 607

  - as formal parameters 609

parameters 100

  - actual 128
  - array elements 134
  - effectively `final` 400
  - `final` 135, 232, 400
  - fixed arity 136
  - formal *see* formal parameters
  - implicit 106
  - main method 141
  - passing 127
  - primitives 129
  - program 141
  - references 130
  - `this` 106
  - variable arity 136

parent class 191
parent directory (`..`) 1289
parent resource bundle 1108
parentheses 50
`ParseException` 371
parsing 1115

  - compact number form 1121
  - currency values 1119
  - date and time 1127
  - numbers 1119
  - percentage values 1120
  - using patterns 1150

parsing numeric values 434
partial application 723
partial implementation 218, 242

partly synchronizing a method 1391
pass by value 128
passing
  parameters 127
  references 130
  variable arity parameter 136
passwords
  reading from console 1256
path 1288
  comparing with another path 1303
  constructing relative path to another path 1300
  converting 1297
  converting to a real path 1302
  converting to an absolute path 1298
  creating 1290, 1291
  interoperability with `File` class 1292
  interoperability with `URI` class 1293
  name elements 1288
  name separator 1288, 1291
  naming conventions 1288
  normalizing 1299
  querying 1294
  resolving with another path 1299
`Path` interface 1287
  see also path
paths
  see class path
paths of execution 1367
path-separator character 340

pattern letters 1127, 1134, 1135
pattern match operator 274
pattern matching 274
pattern symbols 1125
pattern variable 275

- cannot shadow local variable 276
- declare final 276
- scope 275, 276
- shadow a field 276
- patterns
- format elements 1140

percentage 1116, 1117

- formatting 1116, 1119

percentage formatter 1117, 1120
performance 5
period 1057

- compared with duration 1072
- creating 1057
- date-based 1057, 1064
- equality 1060
- get methods 1059
- immutable 1057
- normalization 1061
- parsing 1058
- period-based loop 1062
- plus/minus methods 1061
- supported `ChronoUnit`s 1060
- temporal arithmetic 1061
- text representation 1058, 1067
- thread-safe 1057
- `with` methods 1060

`Period` class 1057

  see period

periodic tasks 1441
`permits` clause 312
permitted direct subclasses 312

  accessibility 314
  contract 313
  extending direct sealed superclass 313
  `final` 313
  locality 314
  `non-sealed` 313
  `sealed` 314

permitted direct subtypes

  contract 316
  enum types 316
  record classes 316

permute elements 860
persistence

  see serialization

plain data 299
plain data class 299
plain data objects 299
plain JAR 1189, 1206
Plain Old Java Objects (POJOs) 299
platform modules 1166
plurals 1146
poison value 1499
POJOs 299
polled locking 1462, 1467
polymorphic method 593

polymorphism 260, 278, 282, 518
polynomial 759
portability 4
Portable Operating System Interface (POSIX) 1325
positional order 644
position-based access 801
positive zero 61
POSIX file permissions

  enum type 1326
  full permissions 1343
  group 1326
  others 1326
  owner 1326
  read (`r`), write (`w`), execute (`x`) 1325
  string format 1326

`PosixFileAttributes` interface 1330, 1332
`PosixFileAttributeView` interface 1334, 1336
`PosixFilePermission` enum type 1325
`PosixFilePermissions` class 1326
postfix operators 50
power 480
precedence rules 51
precision 56
predefined literals 31
predicate 689
`Predicate` functional interface 703

  composing predicates 704
  primitive type specializations 706

preempted threads 1386
preemptive scheduling 1386
prefix 33

  - `0` 33
  - `0x` 33

prepared statement 1526

  - substitutional parameterization 1526

primitive data types 41

  - *see* primitive types

primitive types 44

  - autoboxing 430
  - unboxing 432
  - *see also* primitive values

primitive values

  - assignment 54
  - binary representation 1238
  - equality 75
  - passing 129

principle exported package 1171
principle of least privilege (PoLP) 1611
print writers 1245

  - formatted values 1247
  - text representation of values 1246

printing values 24
`PrintWriter` class 1244
priorities 1385
priority

  - priority queue 816

priority heap 788, 816
priority of thread 1385

  - starvation 1412

- priority queue 787, 788, 815
- `PriorityBlockingQueue` 1496
- `PriorityQueue` 815
- `private` members 351
- privileged code 1609
- privileges 1609
- PRNG
  - *see* pseudorandom number generators 483
- procedural programming paradigm 3
- process of elimination 1619
- process-based multitasking 1367
- processes 1367
- program
  - application 19
  - arguments 141
  - command line 141
  - compiling 20
  - correctness 1385
  - formatting 39
  - performance 1385
  - running 21
  - single-file source-code 21
- program arguments 141
- program counter (PC) 1370
- program output 24
- programmatic locking 1460
  - interruptible locking 1462, 1467
  - lock acquisition 1460
  - lock disciplines 1460

  - polled locking 1462, 1467
  - read-write locking 1466
  - reentrant 1465
  - reentrant locking 1461
  - timed locking 1462, 1467
  - unconditional locking 1467

- programming to interfaces 647, 1196
- proleptic year 1029
- promotion 49
- proper prefix 871, 872
- properties 6
  - *see also* class members
- properties file
  - comment 1101
  - escaping whitespace 1101
  - key–value pair syntax 1100
  - key–value pairs 1100
  - properties 1100
  - property list 1100
  - text file 1101
  - *see also* `Properties` table
  - *see also* property resource file
- `Properties` table 1102
- property
  - key–value pair 1100
  - *see also* properties file
  - *see also* `Properties` table
  - *see also* property resource file
- property list 1100
  - *see also* properties file

  see also property resource file

property resource file 1103

  `.properties` file extension 1103
  naming scheme 1103
  properties 1103
  see also properties file
  see also resource bundle

`PropertyResourceBundle` 1105, 1108
`protected` members 350
`provides-with` directive 1178, 1200
pseudorandom generator 1422
pseudorandom number generators 900

  see `Math` class
  see `Random` class

pseudorandom numbers 482, 483
`public` 19
public API 1163, 1170, 1174
`public` members 350
punctuators 32

Q

qualified class instance creation expression 502, 503
qualified export 1176

  see `exports-to` directive

qualified name 251, 492, 496, 497, 500, 501
qualified opens directive

  see `opens-to` directive

qualified superclass constructor invocation 509
qualified `this` 504, 515

query results 1533
`Queue` 814
queue 785, 788

    capacity-restricted 815
    FIFO 821
    head 814
    `LinkedList` 815
    `PriorityQueue` 815
    `Queue` 814
    unbounded 815

quotation mark 38, 39
quotient 62

R

race conditions 1014, 1389
radix

    prefix 33
    see base 431

ragged arrays 124
raised to power 480
random access by index 802
`Random` class 482, 1422
random numbers 482
random order 860
range

    character values 42
    floating-point values 42
    integer values 42

range of values for date fields 1029
range of values for time fields 1028

range-view 656, 801, 810, 811, 846, 1486
ranking criteria 644
raw types 571, 575, 860
    unchecked warnings 575
read lock 1466
read operation 541
readability 1173
readable 1170
readers 1241
Read-Evaluate-Print Loop (REPL) 23
reading binary values 1238
reading bytes 1234
reading characters 1249, 1250
reading data 1233
`ReadLock` 1466
read-write lock
    fairness policy 1470
    read lock 1466
    reentrant 1470
    write lock 1466
read-write locking 1466
realization 244
reclaiming memory 533
record classes 168, 299, 300, 882
    augmenting 304
    basic syntax 300
    compact canonical record constructor 304, 306
    component fields 302
    component list 300
    generating methods of the `Object` class 303
    generic 311

  - get methods 300, 302
  - immutable 302
  - implementing interfaces 310
  - implicit canonical record constructor 302
  - local 494, 519
  - member declarations 307
  - nested 311, 492
  - normal canonical record constructor 304, 305
  - normal non-canonical record constructor 305, 307
  - permitted direct subtypes 316
  - permitted in sealed interfaces 311
  - record construction 303
  - record elements 300
  - record header 300
  - record name 300
  - record serialization 310
  - restrictions 304
  - serialization 1263
  - shallowly immutable 302
  - static member 494
  - top-level 494

- record components 300
- record header 300
- record name 300
- recursive type bound 591, 613
- `RecursiveAction` 1448
- `RecursiveTask` 1448
- reduced graph 1167
- reducing complexity 324
- reduction operations 947
- reentrant 1465
- reentrant locking 1461

  - fairness policy 1461
- reentrant synchronization 1390
- `ReentrantLock` 1461
- reference types 99, 102, 195, 491
  - classes 99
  - enum types 287, 288
  - interface types 237, 246
- reference values 8
- reference variables 102
- references 8, 11, 102, 113, 127
  - abstract types 221
  - aliases 76, 130
  - array 118, 121, 260
  - assignment 55
  - casting 49, 269
  - dangling 533
  - downcasting 45
  - dynamic type 196
  - equality 76
  - field 534
  - local 534
  - narrowing conversions 45
  - `null` *see* null reference
  - parameterized types 569
  - passing 130
  - raw types 588
  - reachable 533, 534
  - static type 196
  - `super` 206, 515
  - `this` 106, 515
  - upcasting 45

  - widening conversions 45
- reflection 629, 1165, 1178
- Reflection API 1587
- reflective access 1191
- reifiable types 623
- relational databases 1512
- relational model 1512
- relational operators 74
- relational tables
  - columns 1512
  - constraints 1512
  - rows 1512
- relative adjusters 1040
- relative path 1288
- reliability 5
- reliable configuration 1163
- remainder 62
- remainder operator
  - `%` 62
- remove whitespace 453
- repeatable annotation type 1576
- replacing characters 452
- required annotation element 1562
- `requires` directive 1169, 1170, 1177
  - readability 1173
- `requires static` directive 1177
- `requires transitive` directive 1175, 1177
- reserved keywords 31
- reserved literals
  - `false` 32, 37

  `null` *see* null reference
  `true` 32, 37

resizable arrays 787, 788, 802
resource

  see resource bundle

resource bundle 1102

  base name 1102
  candidate bundle name 1108
  default resource bundle 1102, 1103, 1109, 1110
  fallback 1110
  key-based lookup 1104
  key–value pairs 1102
  locating resources 1107
  naming convention 1102
  parent resource bundle 1108
  property resource file 1103
  resource 1102
  resource bundle class 1103
  result bundle 1108

resource bundle class 1103
resource bundle family 1102

  see also resource bundle

resource declaration statement 410
resource leakage 407
resource variable

  `final` 410

`ResourceBundle` 1102, 1103
resources 407, 536
restricted type name `var` 100

result bundle 1108

result set 1533

  - concurrency 1539, 1540, 1541
  - concurrency mode 1541
  - cursor 1534
  - customizing 1539
  - extracting column values 1536
  - fetch size 1536
  - holdability 1539, 1540, 1541
  - holdable 1540
  - metadata 1543
  - navigational direction 1539, 1540
  - scrollable 1539
  - type 1539, 1540
  - updatability 1539, 1540

retaining elements 791

retention policy values for `@Retention` 1568

rethrowing exceptions 401

`return` statement 184

  - `@return` tag 184

return type

  - covariant

return value 10

reuse of code 191, 195

reverse DNS 327, 1171

reverse-order key set 847

reverse-order view 847

rhyming order 771

right associativity 52

rightmost bit 34

ripple effect 195

robustness 5, 416, 1607

role relationship 195

root

  see inheritance hierarchy

root component 1287

root node 1287

`round` method 479

rounding functions 478

rounding mode 1122

rounding numbers 479

round-robin scheduling 1386

rows 1512

`Runnable` functional interface 1370, 1371, 1376, 1424

`RUNNABLE` state 1382

Running 1203

running a Java application 21

running from a modular JAR 1191

running from a runtime image 1205

runtime

  bounds checking 121

runtime checks 48, 648

`Runtime` class 425, 537

runtime class 426

runtime constant pool 1370

runtime data areas 1369

runtime environment 533, 1100

runtime image 3, 1204

runtime stack

  see JVM stack

runtime type

- *see* dynamic type

`RuntimeException` 372, 397

S

scheduled executor service

- scheduling one-shot tasks 1440
- scheduling periodic tasks 1441

scheduled thread pool 1427

`ScheduledExecutorService` 1440

schedulers 1386

schema 1293

scientific notation 36

scope 352

- block 354
- `catch` clause 381, 410
- class 352
- disjoint 355

scope of local variables 324

sealed classes 312

- direct sealed superclass 313
- in compilation unit 317
- `permits` clause 312
- permitted direct subclasses 312, 313

sealed interfaces

- direct sealed superinterface 315
- in compilation unit 317
- `permits` clause 315
- permitted direct subtypes 315

searching 783, 856

    arrays 866
    collections 861

searching in string 451
searching in string builder 466
secure 5
secure coding 1600
security guidelines 1610
security policies 1608
security threats 1602
`SecurityManager` class 425
seed 484
`SELECT` statement 1515
selection statements 152
selector expression 160
self-reference 146
self-referential data structure 566
semantic definition 30
semicolon 101
sensitive information leakage 1603
separators 32, 51
sequence 784
sequential I/O 1233
sequential streams 891

    execution 958
    extracting lines from a `String` 902
    from a `BufferedReader` 902
    from a `CharSequence` 901
    from arrays 898
    from collections 897
    from factory methods in the `Files` class 903

  - ordered 891
  - unordered 891
- serial access 1478
- serialization 1261
  - customizing 1268
  - inheritance 1270
  - `serialVersionUID` 1275
  - transient fields 1267
  - versioning 1273
- `serialVersionUID` 1275
- service 1178, 1196
- service consumer 1178, 1197, 1202
- service interface 1196, 1197
- service loader 1197, 1200
- service locator 1178, 1197, 1200
- service provider 1178, 1197, 1199
- service provider interface
  - *see* service interface
- `ServiceLoader` class 1197
- `ServiceProvider.Provider` interface 1200
- services 324
- `Set` 804
- set 754, 784
  - capacity 807
  - `HashSet` 806
  - `LinkedHashSet` 806
  - load factor 807
  - `NavigableSet` 810
  - `Set` 804
  - `SortedSet` 810

  - `TreeSet` 812
  - unmodifiable 804, 805
  - unmodifiable views 856

- set methods 358
- set operation 586
- setter methods *see* set methods
- SHA-256 digest algorithm 1603
- shadowing fields by local variables 114
- shadowing local variables 514
- shadowing members 685
- shallow copying 426
- shallowly immutable 302
- shared field 1453
- shared memory 1369
- shift
  - compound operators 90
  - distance 86
  - left 86
  - operators 86
- shifting 86
- `short` 32, 42
- short cut
  - *see* symbolic link
- short-circuit 80
  - evaluation 80
- short-circuit execution 908, 947, 952
- shuffle elements 860
- sign bit 86
- sign fill 86
- signature 108, 202, 616

signed shift 86

simple

  assignment operator 54

  `if` 152

  statement 101

simple name 345

simple type name 326

`SimpleFormat` 1115

simplicity 4

single implementation inheritance 193

single inheritance of implementation 193

single quote (') 37, 38

single source file 22

single static import 331

single thread executor 1426

single thread scheduled executor 1427

single-element annotation 1563, 1565

  array-valued single-element annotation 1566

  optional annotation element 1566

  single-element array-valued single-element annotation 1566

  `value()` element 1565

single-element annotation type 1560

single-file source-code program 21

single-line comment 7, 40

singleton 1256

single-valued maps 830

sink nodes 1172

skeletal source file 325

skip list 1492

sleeping 1395

software security 1600
sorted map 785

    comparator 845

sorted set 784, 810, 811

    comparator 811
    half-open interval 810

`SortedMap` 845
`SortedSet` 810
sorting 783, 856, 929
sorting arrays 865
sorting lists 858
source

    file 20, 328
    file name 325
    file structure 325

source directory 1219
source of data 1233
SP *see* space
space 38
spaces 39
spawning threads 1378
special character values 38
specialization 13
specialized classes 194
spurious wakeup 1399
SQL 1514
SQL injection 1525, 1605
SQL statement API

    basic statement 1523
    callable statements 1530

    prepared statement 1526

SQL statements 1522
`SQLException` 371, 1518
square root 480
stability guarantee 911, 930
stack

    generic 598
    LIFO deque 821

stack frame

    see method execution

stack trace 367, 370, 415

    see method execution

`StackOverflowError` 374
stacks 815
standard error stream 370, 1255
standard input stream 1255
standard modules 1166

    JMOD archive 1166

standard out 24
standard output stream 1255
standard time 1082
`StandardCopyOption` enum type 1308
starting threads 1371
starvation 1412
state *see* object state
stateful operations 908
stateless behavioral parameters 910
stateless operations 908
statement block 682

statement terminator 1514

statements 101

- `break` 180
- compound 101
- conditional 152
- `continue` 182
- control flow 101
- control transfer 179
- declaration 65, 70, 81, 101
- `do-while` 173
- empty 101
- expression 70, 101
- `for(;;)` 174
- `for(:)` 176
- `if` 152
- `if-else` 153
- iteration 172
- labeled 179
- resource declaration 410
- `return` 184
- selection 152
- simple 101
- simple `if` 152
- `switch` statement 155, 160
- `synchronized` 1391, 1392
- `throw` 386
- transfer 179
- `try` 377
- `try`-with-resources 407
- `while` 172
- `yield` 164

static

    members *see* static members
    methods 11, 13, 101
    variable initialization 103
    variables *see* static variables

`static` 10, 251, 331
static code

    see static context 99, 100

static context 99, 100
static factory method 359, 1116
static field initializers 99
static fields 13, 113
static import 331

    conflicts 334
    on demand 331
    shadow static members 333
    single static import 331

static initializer blocks 99, 545

    declaration order 545
    exception handling 549
    local variable type inference 545, 547

`static` keyword 113, 545

    static fields 113
    static methods 115

static local enum types 492, 494
static local interfaces 492, 494
static local nested types

    see static local types 491

static local record classes 492, 494

static local types 491

- enum types 492, 494
- interfaces 492, 494
- record classes 492, 494
- *see* local enum types
- *see* local interfaces
- *see* local record class

static member classes 494, 495

- class file name 497
- declaration 495
- direct access 498
- fully qualified name 497
- instantiation 497
- qualified name 496
- UML class diagram 498

static member enum types 494
static member interfaces 494
static member record classes 494
static member types 491

- static member class 491
- static member enum type 492
- static member interface 491
- static member record class 492

`static` members 10, 11, 13
static method references 726
static methods 115

- hiding from superclass 203

static nested classes

- *see* static member types

static nested enum types

- *see* static member enum types

static nested interfaces

- *see* static member interfaces

static nested record class

- *see* static member record class

static nested types

- *see* static member types

static type 196
`static` variables 11, 13, 106
stored functions 1530

- IN parameter 1530
- INOUT parameter 1532
- OUT parameter 1530

stored procedures 1530

- IN parameter 1530
- INOUT parameter 1532
- OUT parameter 1530

stream 798

- transformations 905

Stream API 889

- `BaseStream` interface 889
- building streams 890
- `Collector` interface 890
- `Collectors` 890
- `DoubleStream` interface 889
- `IntStream` interface 889

  - `LongStream` interface 889
  - `Stream` interface 889
  - `Stream` interface 889

stream mapping 906

stream operations 881

  - non-interfering behavioral parameters 910
  - stateless behavioral parameters 910

stream pipelines

  - composing 886
  - executing 887

streams 881

  - *see also* I/O streams
  - aggregate operations 881
  - building from a `BufferedReader` 902
  - building from a `CharSequence` 901
  - building from a range 898
  - building from arrays 898
  - building from collections 897
  - building from directory entries 1345
  - building from specified values 893
  - building from the `Random` class 900
  - building using factory methods in the `Files` class 903
  - collecting into a list 972
  - collecting into an array 971
  - compare with collections 888
  - concatenating 895
  - conversion 934
  - counting elements 953
  - data source 884
  - dropping elements 910, 913

empty 893
encounter order 891
examining elements 920
execution mode 933
extracting lines from a `String` 902
filtering 910
finding first/any 952
finding min/max 954
finite 892
flattening 924
functional reduction 953, 954, 955, 964, 972
infinite 892
intermediate operations 885, 905, 908
lazy execution 907
mapping 905, 921
map-reduce transformations 905
matching operations 949
method chaining 906
mutable reduction 947, 964
see also collectors
numeric 885, 892
numeric summary statistics 972
object 885, 892
operations with side effects 946, 948
ordered 891
parallel streams 891, 962, 966, 1009
reduction operations 947, 964
replacing each element with multiple elements 927
searching operations 947
selecting distinct elements 915
sequential 891
sequential execution 958
setting as unordered 932

  - short-circuit execution 908, 947, 952
  - skipping elements 915
  - sorting 929
  - stability guarantee 911, 930
  - stateful operations 908
  - stateless operations 908
  - Stream API 889
  - stream mapping 906
  - stream operations 881
  - stream pipelines 886
  - taking elements 910, 913
  - terminal stream operations 885, 946
  - truncating 917
  - unordered 891

`strictfp` 58

string builders 69

  - appending 467
  - capacity 464, 470
  - `compareTo` 467
  - construction 465
  - copying characters 465
  - deleting 468
  - extracting substrings 466
  - individual characters 465
  - inserting 468
  - joining 450
  - length 465
  - mutable 464
  - reading characters 465
  - replacing 468
  - reversing 469
  - searching 466

  - setting length 470
  - substrings 466
  - trimming 470

`String` class

  - *see* strings

string conversion 46, 68, 455

string literal pool 440

  - interned 440

string literals 440

  - `case` labels 162
  - hash value 162
  - interned 440

string pattern 1125

`StringBuffer` legacy class

  - thread-safe 464

`StringBuilder` class

  - *see* string builders

strings

  - capacity 470
  - changing case 448
  - compact 440
  - `compareTo` 447
  - comparing 444, 447, 761
  - concatenation 67, 448
  - concatenation operator `+` 69
  - contains 452
  - conversions 455
  - convert to character array 445

copying characters 445
creating 440
empty 441, 444
ends with 452
`equals` 447
extracting lines 902
extracting substrings 453
finding index 451
formatted 457
ignoring case in comparison 447
immutable 440
indenting lines 456
individual characters 445, 465
initializing 440
integer stream 444
internal representation 439
interned 440, 442
joining 450, 843, 984
length 444, 445
lexicographical ordering 447
literals 39, 440
natural order 761
read character at index 444, 445
repeating 449
replacing 452
searching 451
selector expression 156
starts with 452
streams 901, 902
string literal pool 440
stripping 453
substrings 453
text blocks 458

    thread-safe 440
    transforming 455
    trimming 453

strongly typed language 48
Structured Query Language 1514
style-based formatter for dates and times 1131
subclass 13, 15, 191
subdirectories 1287
subformat pattern 1146
subinterface 244
sublist 656
submap view 845, 846
sub-maps 1491
subsequence 444
subset 804, 808, 810, 811, 1486
subsignature 617
substitutional parameterization 1526
substring searching 451, 466
substrings 451, 453, 466
subtags 1098
subtype bivariance 581
subtype contravariance 581
subtype covariance 259, 260, 580, 581
subtype invariance 581
subtype notation `<:` 584
subtype relationship 648
subtypes 242, 278, 581, 592
subtype–supertype relationship 45, 195
suffix

    `D` 36
    `F` 36
    `L` 32

`super` 248

  - construct 211
  - keyword 201, 206
  - reference 206

superclass 13, 191
superclass constructor 110
superinterfaces 244
superset 808
supertypes 242, 278, 582
`Supplier` functional interface 699

  - primitive type specializations 700

suppressed exceptions 370, 415
suppressed warnings

  - deprecation warnings 1583
  - unchecked warnings 1583

switch block

  - local variable scope 169
  - switch rules 160

`switch` expression 164

  - arrow notation 166
  - `case` labels 164
  - colon notation 164
  - comparison with `switch` statement 169
  - `default` clause 166
  - exhaustive 165, 166, 167
  - fall-through 165
  - local variable scope 169
  - return multiple values 168
  - `yield` statement 164, 166

switch rules 160

- mutually exclusive 161

`switch` statement 155

- arrow notation 160
- `break` 157
- `case` labels 156
- colon notation 155, 157
- comparison with `switch` expression 169
- `default` clause 156, 161
- enum types 163
- execution 156
- exhaustive 163
- fall-through 156
- local variable scope 169
- selector expression 160
- using strings 162

switching between threads 1368
symbolic link 1289, 1304
synchronization 1387
`synchronized`

- keyword 1388
- methods 1388
- starvation 1412
- statements 1391, 1392
- vs. `volatile` 1456

synchronized block 1391
synchronized collections 1477

- compound mutually exclusive operations 1480
- serial access 1478
- underlying collection 1477

synchronized method 1388
synchronized views of collections
  see synchronized collections

`SynchronousQueue` 1496
syntactically legal 30
`System` 425
  `out` 24
  reassigning standard streams 1255
  standard streams 1255

system clock 1030
system default zone ID 1074
system modules 1212

T

TAB *see* horizontal tab
table definition 1513
tabs 39
tabulators 39
tagging interface
  see marker interfaces

tags 40
tail map view 845, 846
tail set view 810, 811, 1486
taking stream elements 910, 913
target directory 1187
target parallelism level 1426
target reference 724
target type 570, 682
target typing 684
task 1367, 1423

task cancellation 1444

task execution policy 1426

TCL 1514

telephone directory order 447

temporal arithmetic 1040, 1054, 1061, 1069

temporal fields 1027

temporal objects

- access specific temporal values 1032, 1077
- before/after methods 1034
- combining date and time 1030
- common methods 1026
- comparing 1034
- conversion to `Date` 1088
- creating 1027, 1075
- date 1024
- date fields 1024
- date units 1040
- date-time 1024
- fields 1024, 1025
- formatting 1127
- `get` methods 1032, 1077
- immutable 1024, 1075
- method naming convention 1026
- parsing 1127
- plus/minus methods 1040, 1081
- range of values for date fields 1029
- range of values for time fields 1028
- temporal arithmetic 1040, 1061, 1062, 1071, 1080
- temporal fields 1027, 1036
- thread-safe 1024, 1075
- time 1024
- time fields 1024

  - time units 1040
  - `with` methods 1035, 1078
  - zoned date-time 1024, 1074
- `TemporalAmount` interface 1061
  - *see also* `Duration`
  - *see also* `Period`
- `TemporalField` interface
  - *see also* `ChronoField`
- `TemporalUnit` interface
  - *see also* `ChronoUnit`
- terminal stream operations 885, 946
  - averaging 974
  - collecting into a list 972
  - collecting into an array 971
  - counting elements 953
  - eager execution 887
  - finding first/any 947, 952
  - finding min/max 954
  - functional reduction 947, 953, 954, 955, 972
    - accumulator 956, 957, 962
    - combiner 956, 962
    - identity value 956, 957, 962
    - parallel execution 962
    - sequential execution 958
  - matching operations 947, 949
  - mutable reduction 947, 964, 971, 972
    - accumulator 965
    - combiner 965

            parallel stream execution 966
            sequential stream execution 965
            supplier 965
            see also collectors

    operations with side effects 946, 948
    reduction operations 947, 964

            see also collectors

    searching operations 947
    short-circuit operations 947, 952
    summary statistics 972, 974
    summation 973

`TERMINATED` state 1382, 1405
terminating loops 180
ternary conditional expressions

    see also conditional expressions 90

ternary conditional operator 50, 90
text block 458

    closing delimiter 458
    content 458
    escape sequences 459
    essential whitespace 462
    incidental whitespace 462
    indentation 462
    opening delimiter 458

text files

    reading from 1249
    stream to read lines 1346
    writing to 1247

text representation 426

`this`
  - reference 106

`this` keyword
  - qualified form 504
  - with `new` 503

`this()` constructor call 209, 555
thousands separator 1125
`Thread` class 1370
thread coordination 1396
thread execution
  - runtime organization 1369

thread interference 1373, 1389
thread interruption rule 1415
thread lifecycle 1380
thread of execution 1367
thread pool 1423, 1425
thread safety 1368
Thread start rule 1414
Thread termination rule 1414
thread-based multitasking 1367
thread-local variable 1453
`ThreadLocalRandom` 1422
threads 5, 425, 534, 1402
  - acquire lock 1388
  - alive 1377, 1383
  - `BLOCKED` state 1382
  - blocking operation 1405
  - child 1373, 1378
  - class 1370
  - code executed 1371

controlling termination 1405
creation 1370
current thread 1372
daemon 1377
deadlocks 1408
death 367
deprecated methods 1408
determining state 1381
entry set for 1380
exception propagation 367
extending `Thread` 1375
fairness 1408
happens-before relationship 1414
`IllegalThreadStateException` 1378
interrupt status 1393
interrupted 1399
`InterruptedException` 1393
joining 1383, 1402
JVM stack 534
live 534
livelock 1410
liveness 1408
main thread 1373, 1378
names 1372
`NEW` state 1382, 1386
non-observable substate 1382
non-runnable state 1383
normal 1377
notification 1396, 1398
notifying 427, 1396
Object lock rule 1414
priorities 1385
priority 1383

`run()` method 1372
`Runnable` 1370, 1371, 1376
`RUNNABLE` state 1382
running 1375
running thread 1383
scheduler 1385
shared field 1453
shared memory 1369
sleeping 1383, 1395
spawning 1378
spurious wakeup 1399
`start()` method 1372
starting 1371, 1375, 1386
starvation 1412
states 1381
switching 1368
synchronization 1387
`synchronized` 1388
synchronized method 1388
tasks 1423
`TERMINATED` state 1382, 1405
termination 1405
`Thread` class 1370
thread coordination 1396
Thread interruption rule 1415
thread lifecycle 1380
Thread start rule 1414
thread status 1378
Thread termination rule 1414
thread-local 1453
`TIMED_WAITING` state 1382
transitions 1381
Volatile field rule 1414, 1454

    - wait set for 1397
    - waiting 427, 1396
    - `WAITING` state 1382
    - yielding 1383, 1386
- thread-safe 464, 645, 789, 1024, 1075
    - immutable classes 356
- thread-safe code 1387, 1451
- thread-safe collection 1477
- thread-safety 356
    - atomic variables 1456
    - immutability 1453
    - intrinsic locking 1460
    - programmatic locking 1460
    - synchronized collections 1477
    - unmodifiable collections 1477
    - volatile fields 1453
- `throw` statement 386
- `Throwable` 368, 405, 425
- throw-and-catch paradigm 365
- throwing exceptions 365
- `throws` clause 388
    - and rethrowing 402
- time fields 1024
- time gap 1082
- time overlap 1082
- time standard 1073
- time units 1040, 1421
- time zone 1024
    - local time 1073
    - *see* zone ID

time zone offset 1073
time zone rules 1074, 1082, 1084
timed locking 1462, 1467
`TIMED_WAITING` state 1382
timeline 1024
  - instant 1074
  - local 1074

time-sliced scheduling 1386
timestamps 1049
`TimeUnit` 1421
`ToDoubleBiFunction` interface 718
`ToDoubleFunction` interface 712
`ToIntBiFunction` interface 718
`ToIntFunction` interface 712
tokens 30
`ToLongBiFunction` interface 718
`ToLongFunction` interface 712
top-level classes 491, 494
top-level enum types 491, 494
top-level interfaces 491, 494
top-level record classes 491, 494
top-level types 491
`toString` method 426, 431, 432
total order 761
total ordering 865
Transaction Control Language (TCL) 1514
transactions 1545
  - auto-commit mode 1545
  - commit 1545
  - rollback 1545
  - savepoint 1545

transfer statements 179

`TransferQueue` 1496, 1498

transforming strings 455

transient fields 1267

transitive relation 194

transitive relationship 1414

tree 1287

- traversal 1347

tree map 847

tree set 812, 847

`TreeMap` 847

trees 788

`TreeSet` 812

`trim` method 453

`true` literal 32, 37

truth-values 37, 43

`try` block 377

`try-catch-finally` construct 375

`try`-with-resources 147, 407, 1519

- basic 410
- concise syntax 411
- extended 410
- header 410
- `var` in header 410

tuples 1512

two’s complement 33, 34, 86

two-arity 695

type annotation 1572

type cast 48

type cast expression 269

type checking 682

type comparison operator 270
type contexts 1572
type declarations 325

 nested types 491
 top-level types 491, 494

type erasure 576, 613, 622
type hierarchy 45, 195
type import

 see `import`

type inference 142
type information 565
type instance 569
type parameter constraints 591
type parameters 633

 actual 569
 bounds 591
 formal 567
 recursive bounds 611
 restrictions 568

type pattern 275
type variables 567
type witness 594, 595, 928
types

 arrays 579
 `boolean` 41, 43
 `byte` 32, 42
 casting 48
 `char` 42
 classes *see* classes
 comparison 270

  - compatibility 48
  - `double` 42
  - exceptions 368
  - `float` 42
  - floating-point 41, 42
  - `int` 32, 42
  - integers 41
  - integral types 41
  - `long` 32, 42
  - numeric types 41
  - parsing 434
  - `short` 32, 42
  - unrelated 570
  - wrappers 429
  - *see also* classes
- type-safe 264
- type-safety 646, 648
- *type*`Value` method 432, 434

U

- UML 6
  - accessibility 348
  - aggregation 16
  - classes 6
  - inheritance 14
  - *see also* diagrams
- UML class diagram
  - non-static member classes 508
  - static member classes 498
- UML notation for members 348

UN M.49 three-digit area code 1096
unary arithmetic operators 61
unary numeric promotion 49
unary operators 50
`UnaryOperator` interface 720
  composing unary operators 720
  primitive type specializations 721
unbounded collections 1483
unbounded instance method references 729
unbounded type parameter 581, 583
unbounded type references 590
unbounded wildcard
  `?` 581, 582
unbounded wildcard references 588
unboxing 432
  `do-while` statement 173
  `for(;;)` statement 174
  `for(:)` statement 177
  `if` statement 152
  `if-else` statement 153
  `switch` statement 156
  `while` statement 173
unboxing conversions 45
uncaught exceptions 367
unchecked 627, 630
unchecked array creation warning 630
unchecked call warning 576, 588
unchecked cast warning 625, 627
unchecked conversion warning 571, 576, 585, 627, 646
unchecked conversions 46

unchecked exceptions 374

unchecked generic array creation warning 630, 650, 659, 805, 833, 863

unchecked warnings 45, 575, 576, 1583

- `@SuppressWarnings("unchecked")` 627
- call warning 576, 588
- cast warning 625
- conversion warning 576
- generic array creation warning 630
- heap pollution warning 630, 1585

unchecked generic array warning 1585

unconditional locking 1467

underflow 34, 61

underlying collection 1477

uni-`catch` clause 376, 397

- effectively `final` parameter 400

Unicode 37, 42, 437, 447, 1241

Unicode character encoding 1101

Unicode locale/language extension 1118

unidirectional edges 1166

Unified Modeling Language *see* UML

Uniform Resource Identifier (URI) 1290

union 804

union of alternatives 399

unmodifiable 660

unmodifiable collections 649, 805, 857, 1477

unmodifiable lists 649, 660, 801

unmodifiable maps 832

unmodifiable sets 805

unmodifiable view of a collection 856, 857

unmodifiable view of a list 856

unmodifiable view of a map 857
unmodifiable view of a navigable map 857
unmodifiable view of a navigable set 856
unmodifiable view of a set 856
unmodifiable view of a sorted map 857
unmodifiable view of a sorted set 856
unmodifiable views 856, 857
unmodifiable views of collections 1477
unnamed module 314, 1206
unnamed package 314
unordered streams 891

  from the `Random` class 900

unqualified export 1178

  see `exports` directive

unreachable code 381, 401
unsafe casts 271
unsigned integer 435
unsupported operations 790, 831
`UnsupportedOperationException` 373, 794, 797
`UnsupportedTemporalTypeException` 1033, 1062, 1068, 1070, 1075, 1131
unused keywords 31
upcasting 45
updatability 1539
update expression 174
`UPDATE` statement 1515
upper bound 581, 591
upper bounded wildcard

  `? extends T` 581

upper bounded wildcard references 589
`URI` class 1290, 1293

    schema 1293

user threads

    see normal thread

`uses` directive 1178, 1202
using arrays 120
using packages 329
using variables 102
UTC (Coordinated Universal Time) 1073
UTC/Greenwich 1073

    `Z` 1073

UTF-16 encoding scheme 439, 440
utility class 115, 251, 360

V

value collection 831
value obfuscation 1603
value objects 785
`value()` element 1560, 1561, 1565
`valueOf` method 430, 431, 455
values 34, 830

    constants 32
    overflow 34
    underflow 34
    wrap-around 34
    see also variables

`var` 571
`var` declaration

    see local variable type inference

varargs

- variable arity 136

varargs method

- *see* variable arity method

varargs parameter

- *see* variable arity parameter

variable arity argument 659, 863
variable arity call 139
variable arity method 136
variable arity parameter 136, 630, 863, 1304

- generic array creation 630
- heap pollution 630

variable capture 687
variable declarations 102, 354
variable initialization 11, 102, 104
variables 8, 102

- blank final 135
- blank `final` local 231, 232
- constant values 229
- default values 103
- effectively `final` 687
- `final` 229
- identifiers 102
- in interfaces 254
- initialization *see* variable initialization
- lifetime 106
- local 142, 354
- parameters 100, 128
- reference variable 102
- references 102
- `static` 11

var-type inferred parameters 680

`Vector` 802

vector 802

Venn diagrams 791

versioning 1273

view of a list 656, 660

viewing dependencies 1214

- class-level dependencies 1216
- module-level dependencies 1216
- package-level dependencies 1214

views 810, 839

virtual method invocation

- *see* dynamic method lookup

Virtual Private Network (VPN) 1601

`VirtualMachineError` 374

visibility 1454

visibility modifiers

- *see* access modifiers

`void` 19, 184, 429

`void` return 681

`Void` wrapper class 429

`volatile`

- vs. `synchronized` 1456

Volatile field rule 1414, 1454

volatile fields 1453

- visibility 1454

volume name 1287

W

wait set 1396, 1397

  starvation 1412

waiting 1396

  timeout 1399

`WAITING` state 1382
waiting threads 427
`WHERE` clause 1515, 1516
`while` statement 172
whitespace 39, 453, 454
whole–part relationship 194
widening conversions

  primitive 44
  references 45

widening reference conversions 196, 270, 584
wider range 44
wildcard parameterized types 584

  type hierarchy 581, 582, 583

wildcard types 580

  see wildcards

wildcards 579, 580

  `?` 588
  `? extends T` 589
  `? super T` 589
  assignment 584
  bounded 581
  capture 604
  capture conversion 606
  nested 607

    parameterized types 584
    restrictions 584
    subtype relationships 579
    type hierarchy 581, 582, 583

withers 1035

work stealing pool 1426

    target parallelism level 1426

worker threads 1423

wrapper 1233

wrapper classes 41, 425, 426, 429, 744, 754, 761

    interned values 433

wrapper type 45, 46, 59, 70

write lock 1466

write operation 542

`WriteLock` 1466

writers 1241

writing binary values 1238

writing bytes 1234

writing characters 1238, 1249, 1253

writing data 1233

X

xor 83

Y

`yield` statement 164, 166

yielding 1386

Z

`Z` 1073
zero
  negative 61
  positive 61
zero fill shift 86
zero-arity 695
zero-based index 644, 801
zone ID 1073
  available zone IDs 1073
  system default 1074
zoned date-time
  accessing time zone 1077
  accessing time zone offset 1077
  conversion to `Date` 1088
  converting to different time zone 1079
  converting to instant 1078
  see also daylight saving time (DST)
  see temporal objects
`ZonedDateTime` class 1072, 1074
  see temporal objects
  see zoned date-time
`ZoneId` class 1072
`ZoneOffset` class 1072

#1 programming language for today's tech trends

Java™

www.oracle.com/java

Java magazine

Take your skills to the next level with *Java Magazine*

Written for developers by developers, new articles are posted every two weeks, and there's a fresh quiz every Tuesday!

Read all the articles and subscribe to the newsletter at no cost.
oracle.com/javamagazine

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
// ...
doSomethingInteresting();                                   // (1)
// ...
```

https://www.mughal.no/jse17ocp/

www.oracle.com/technetwork/java/javase/overview/index.html http://docs.oracle.com/javase/specs/index.html

khalid@mughal.no
vasiliy.a.strelnikov@oracle.com

```
// File: Point2D.java
public class Point2D {             // Class name
  // Class Member Declarations

  // Fields:                                                        (1)
  private int x;     // The x-coordinate
  private int y;     // The y-coordinate

  // Constructor:                                                   (2)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Methods:                                                       (3)
  public int  getX()           { return x; }
  public int  getY()           { return y; }
  public void setX(int xCoord) { x = xCoord; }
  public void setY(int yCoord) { y = yCoord; }
  public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y)
}
```

```
Point2D p1 = new Point2D(10, 20);  // A point with coordinates (10,20)
```

```
Point2D p2 = new Point2D(5, 15);  // A point with coordinates (5,15)
```

```
Point2D point = new Point2D(-1, -4);   // Creates a point with coordinates (-1,-4)
point.setX(-2);                        // (1) The x field is set to the value -2
int yCoord = point.getY();             // (2) Returns the value -4 of the y field
System.out.println(point.toString());  // (3) Prints: (-2,-4)
point.distanceFromOrigin();            // (4) Compile-time error: No such method.
```

```
    System.out.println(point.x);        // (1) Compile-time error: x is not accessible.
    System.out.println(point.getX());  // OK.
```

```
private static String info = "A point represented by (x,y)-coordinates.";
```

```
double d = Point2D.distance(p1, p2); // Class name to invoke static method
```

```
p1.showInfo();                       // Reference invokes static method
```

```
// File: Point2D.java
public class Point2D {             // Class name
  // Class Member Declarations
  // Static variable:                                                 (1)
  private static String info = "A 2D point represented by (x,y)-coordinates.";

  // Instance variables:                                              (2)
  private int x;
  private int y;

  // Constructor:                                                     (3)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Instance methods:                                                (4)
  public int  getX()           { return x; }
  public int  getY()           { return y; }
  public void setX(int xCoord) { x = xCoord; }
  public void setY(int yCoord) { y = yCoord; }
  public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y)

  // Static methods:                                                  (5)
  public static double distance(Point2D p1, Point2D p2) {
    int xDiff = p1.x - p2.x;
    int yDiff = p1.y - p2.y;
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
  }
  public static void showInfo() { System.out.println(info); }
}
```

```
public class Point3D extends Point2D {             // (1) Uses extends clause
  // ...
}
```

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

```
// File: Point3D.java
public class Point3D extends Point2D {           // (1) Uses extends clause

  // Static variable:                                                   (2)
  private static String info = "A 3D point represented by (x,y,z)-coordinates.";

  // Instance variable:                                                 (3)
  private int z;

  // Constructor:                                                       (4)
  public Point3D(int xCoord, int yCoord, int zCoord) {
    super(xCoord, yCoord);                                          // (5)
    z = zCoord;
  }

  // Instance methods:                                                  (6)
  public int  getZ()           { return z; }
  public void setZ(int zCoord) { z = zCoord; }
  @Override
  public String toString() {
    return "(" + getX() + "," + getY() + "," + z + ")"; // Format: (x,y,z)
  }

  // Static methods:                                                    (7)
  public static double distance(Point3D p1, Point3D p2) {
    int xDiff = p1.getX() - p2.getX();
    int yDiff = p1.getY() - p2.getY();
    int zDiff = p1.getZ() - p2.getZ();
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff + zDiff*zDiff);
  }
  public static void showInfo() { System.out.println(info); }
}
```

```
Point3D p3A = new Point3D(10, 20, 30);
System.out.println(p3A.toString());             // (10,20,30)      (Point3D)
System.out.println("x: " + p3A.getX());         // x: 10           (Point2D)
System.out.println("y: " + p3A.getY());         // y: 20           (Point2D)
System.out.println("z: " + p3A.getZ());         // z: 30           (Point3D)
```

```
p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
System.out.println(p3A.toString());              // (-10,-20,-30)    (Point3D)

Point3D p3B = new Point3D(30, 20, 10);
System.out.println(p3B.toString());              // (30,20,10)       (Point3D)
System.out.println(Point3D.distance(p3A, p3B));   // 69.2820323027551 (Point3D)
Point3D.showInfo(); // A 3D point represented by (x,y,z)-coordinates. (Point3D)
```

```
Line line1 = new Line(new Point2D(5,6), new Point2D(7,8));
System.out.println(line1.toString());                  // Line[(5,6),(7,8)]
line1.setEndPoint1(new Point2D(11, 12));
line1.setEndPoint2(new Point2D(13, 14));
System.out.println(line1.toString());                  // Line[(11,12),(13,14)]
System.out.println("Length: " + line1.length());    // Length: 2.8284271247461903
```

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

```
// File: Line.java
public class Line {

  // Instance variables:                                                     (1)
  private Point2D endPoint1;
  private Point2D endPoint2;

  // Constructor:
  public Line(Point2D p1, Point2D p2) {
    endPoint1 = p1;
    endPoint2 = p2;
  }

  // Methods:
  public Point2D getEndPoint1() { return endPoint1; }
  public Point2D getEndPoint2() { return endPoint2; }
  public void setEndPoint1(Point2D p1) { endPoint1 = p1; }
  public void setEndPoint2(Point2D p2) { endPoint2 = p2; }
  public double length() {                                                // (2)
    return Point2D.distance(endPoint1, endPoint2);
  }
  public String toString()  {
    return "Line[" + endPoint1 + "," + endPoint2 + "]";
  }
}
```

```
public class Counter {                                                    // (1)
  int current, step;
  public Counter(int startValue, int stepValue) {                         // (2)
    setCurrent(startValue);                                               // (3)
    setStep(stepValue);
  }
  public int  getCurrent()           { return current; }                  // (4)
  public void setCurrent(int value)  { current = value; }                 // (5)
  public void setStep(int stepValue) { step = stepValue; }                // (6)
}
```

```
Thing item, stuff;
item = new Thing();
Thing entity = new Thing();
```

```
class A {
  protected int value1;
}

class B extends A {
  int value2;
}
```

```
public static void main(String[] args)   // Method header
{ /* Implementation */ }
```

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

```
// File: Point3D.java
public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

```
// File: TestPoint3D.java
public class TestPoint3D {
  public static void main(String[] args) {
    Point3D p3A = new Point3D(10, 20, 30);
    System.out.println("p3A: " + p3A.toString());
    System.out.println("x: " + p3A.getX());
    System.out.println("y: " + p3A.getY());
    System.out.println("z: " + p3A.getZ());
    p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
    System.out.println("p3A: " + p3A.toString());

    Point3D p3B = new Point3D(30, 20, 10);
    System.out.println("p3B: " + p3B.toString());
    System.out.println("Distance between p3A and p3B: " +
                       Point3D.distance(p3A, p3B));
    Point3D.showInfo();
  }
}
```

```
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

```
>javac Point2D.java Point3D.java TestPoint3D.java
```

```
// File: Demo-App.java
public class TestPoint3D {
  // Same as in Example 1.5.
  // Provides the main() method and is the first class declaration in the file.
}

public class Point2D {
  // Same as in Example 1.2.
}

public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

```
>java Demo-App.java
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

```
>jshell
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> int i = 20
i ==> 20

jshell> Math.sqrt(i)
$6 ==> 4.47213595499958

jshell> 3 + 4 * 5
$7 ==> 23

jshell> (3 + 4) * 5
$8 ==> 35

jshell> /exit
|  Goodbye
>
```

```
System.out.println(2022);                              // 2022
```

```
Point2D origin = new Point2D(0, 0);
System.out.println(origin.toString());                      // (0,0)
System.out.println(origin);                                 // (0,0)
System.out.println("Stranger Strings".toString());      // Stranger Strings
System.out.println("Stranger Strings");                 // Stranger Strings
System.out.print("Don’t terminate this line!");
```

```
PrintStream printf(String format, Object... args)
```

```
System.out.printf("Formatted values|%5d|%8.3f|%5s|%n", // Format string
                  2016, Math.PI, "Hi");               // Values to format
```

```
Formatted values| 2016|  3.142|   Hi|
public class SmallProg {
  public static void main(String[] args) { System.out.println("Good luck!"); }
}
```

```
number, Number, sum_$, bingo, $$_100, _007, mål, grüß
```

48chevy, all@hands, grand-sum, _

```
0.0       0.0d       0D
0.49      .49        .49D
49.0      49.        49D
4.9E+1    4.9E+1D    4.9e1d   4900e-2  .49E2
```

```
0.0F      0f
0.49F     .49F
49.0F     49.F       49F
4.9E+1F   4900e-2f   .49E2F
```

```
0b0111_1111_1111_1111_1111_1111_1111_1111
0_377_777_777            0xff_ff_ff_ff
-123_456.00              1_2.345_678e1_2
2009__08__13             49_03_01d
```

```
_0_b_01111111111111111111111111111111_
_0377777777_                _0_x_ffffffff_
+_123456_._00_              _12_._345678_e_12_
_20090813_                  _490301_d_
```

```
"Here comes a tab.\t And here comes another one\u0009!"        (1)
```

```
"Here comes a tab.\t And here comes another one\u0009!"    (1)
"What's on the menu?"                                      (2)
"\"String literals are double-quoted.\""                   (3)
"Left!\nRight!"                                            (4)
"Don't split                                               (5)
me up!"
```

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
Left!
Right!
```

```
// This comment ends at the end of this line.
int age;        // From comment-start sequence to the end of the line is a comment.
```

```
/* Formula for alchemy.
   gold = wizard.makeGold(stone);
   /* But it only works on Sundays. */
*/
```

```
/**
 *  This class implements a gizmo.
 *  @author K.A.M.
 *  @version 4.0
 */
```

```
long year = 2020;    // (1) Implicit widening: long <----- int, assigned 2020L
int pi = (int) 3.14; // (2) Narrowing requires cast: int <----- double, assigned 3
```

```
Object obj = "Upcast me";  // (1) Widening: Object <----- String
```

```
String str = (String) obj; // (2) Narrowing requires cast: String <----- Object
```

```
Integer iRef = 10;                    // (1) Implicit boxing: Integer <----- int
Double dRef = Double.valueOf(3.14);  // (2) Explicit boxing: Double <----- double

int i = iRef;                        // (3) Implicit unboxing: int <----- Integer
double d = dRef.doubleValue();       // (4) Explicit unboxing: double <----- Double
```

```
int i = (int) 10;                    // int <---- int
String str = (String) "Hi";          // String <---- String
```

```
byte b = 10;   // Narrowing conversion: byte <--- int
```

```
// Assignment: (1) Implicit narrowing followed by (2) boxing.
Character space1 = 32;     // Character <-(2)-- char <-(1)-- int

// Invocation of method with signature: valueOf(char)
Character space2 = Character.valueOf(32);       // Compile-time error!
                                                // Call signature: valueOf(int)
Character space3 = Character.valueOf((char)32); // OK!
                                                // Call signature: valueOf(char)
```

```
long l = (long) 10;          // Widening primitive conversion: long <--- int
int i = (int) l;             // (1) Narrowing primitive conversion: int <--- long
Object obj = (Object) "7Up"; // Widening ref conversion: Object <--- String
String str = (String) obj;   // (2) Narrowing ref conversion: String <--- Object
Integer iRef = (Integer) i;  // Boxing: Integer <--- int
i = (int) iRef;              // Unboxing: int <--- Integer
```

```
int b = 10;
System.out.println((b=3) + b);
```

```
(b=3) + b
  3   + b      b is assigned the value 3
  3   + 3
  6
```

```
out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));  // (1)
```

```
import static java.lang.System.out;

public class EvalOrder{
  public static void main(String[] args){

    int j = 2;
    out.println("Evaluation order of operands:");
    out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));    // (1)

    int i = 1;
    out.println("Evaluation order of arguments:");
    add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n")); // (2) Three arguments.
  }

  public static int eval(int operand, String str) {       // (3)
    out.print(operand + str);       // Print int operand and String str.
    return operand;                 // Return int operand.
  }

  public static void add3(int operand1, int operand2, int operand3) {    // (4)
    out.print(operand1 + operand2 + operand3);
  }
}
```

```
Evaluation order of operands:
2 + 3 * 4
14
Evaluation order of arguments:
1, 2, 3
6
```

```
add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n"));  // (2) Three arguments.
```

```
int j, k;
j = 0b10;          // j gets the value 2.
j = 5;             // j gets the value 5. Previous value is overwritten.
k = j;             // k gets the value 5.
```

```
int i;
i = 5;            // i gets the value 5.
i = i + 1;        // i gets the value 6. + has higher precedence than =.
i = 20 - i * 2;   // i gets the value 8: (20 - (i * 2))
```

```
Pizza pizza1 = new Pizza("Hot&Spicy");
Pizza pizza2 = new Pizza("Sweet&Sour");

pizza2 = pizza1;
```

```
int j, k;
j = 10;     // (1) j gets the value 10, which is returned
k = j;      // (2) k gets the value of j, which is 10, and this value is returned
```

```
System.out.println(j = 10);   // j gets the value 10, which is printed.
System.out.println(k = j);    // k gets the value of j, i.e. 10, which is printed
```

```
k = j = 10;      // (k = (j = 10))
```

```
Pizza pizzaOne, pizzaTwo;
pizzaOne = pizzaTwo = new Pizza("Supreme"); // Aliases
```

```
int[] a = {10, 20, 30, 40, 50}; // An array of int (§3.9, p. 119)
int index = 4;
a[index] = index = 2;           // (1)
```

```
a[index] = index = 2;
a[4]     = index = 2;
a[4]     = (index = 2);        // index gets the value 2. = is right associative.
a[4]     =      2;             // The value of a[4] is changed from 50 to 2.
```

```
int v1 = v2 = 2016;             // Only v1 is declared. Compile-time error!
```

```
// Widening Primitive Conversions
int    smallOne = 1234;              // No widening necessary.
long   bigOne   = 2020;              // Widening: int to long.
double largeOne = bigOne;            // Widening: long to double.
double hugeOne  = (double) bigOne;   // Cast redundant but allowed.
```

```
long bigInteger = 98765432112345678L;
float fpNum = bigInteger;  // Widening but loss of precision: 9.8765436E16
```

```
int result = 100;                // Not a constant variable. Not declared final.
final char finalGrade = 'A';     // Constant variable. ’A’
```

```
System.out.printf("%d%n%s%n%d%n%.2f%n%b%n%d%n%d%n",
    2022,                       // Constant expression. 2022
    "Trust " + "me!",           // Constant expression. "Trust me"
    2 + 3 * 4,                  // Constant expression. 14
    Math.PI * Math.PI * 10.0,   // Constant expression. 98.70
    finalGrade == 'A',          // Constant expression. true
    Math.min(2020, 2021),       // Not constant expression. Method call.
    ++result                    // Not constant expression. Increment operator.
);
```

```
// Conditions fulfilled for implicit narrowing primitive conversions.
short s1 = 10;       // int value in range.
short s2 = 'a';      // char value in range.
char c1 = 32;        // int value in range.
char c2 = (byte)35;  // byte value in range. (int value in range, without cast.)
byte b1 = 40;        // int value in range.
byte b2 = (short)40; // short value in range. (int value in range, without cast.)
final int i1 = 20;   // Constant variable
byte b3 = i1;        // final value of i1 in range.
```

```
// Conditions not fulfilled for implicit narrowing primitive conversions.
// A cast is required.
int i2 = -20;            // i2 is not a constant variable. i2 is not final.
final int i3 = i2;       // i3 is not a constant variable, since i2 is not.
final int i4 = 200;      // i4 is a constant variable.
final int i5;            // i5 is not a constant variable.
short s3 = (short) i2;   // Not constant expression.
char  c3 = (char)  i3;   // Final value of i3 not determinable at compile time.
char  c4 = (char)  i2;   // Not constant expression.
byte  b4 = (byte)  128;  // int value not in range.
byte  b5 = (byte)  i4;   // Value of constant variable i4 is not in range.
i5 = 100;                // Initialized at runtime.
short s4 = (short) i5;   // Final value of i5 not determinable at compile time.
```

```
// The value is truncated to fit the size of the target type.
float huge   = (float) 1.7976931348623157d;  // double to float.
long  giant  = (long)  4415961481999.03D;    // (1) double to long.
int   big    = (int)   giant;                // (2) long to int.
short small  = (short) big;                  // (3) int to short.
byte  tiny   = (byte)  small;                // (4) short to byte.
char  symbol = (char)  112.5F;               // (5) float to char.
```

```
Boolean   boolRef = true;  // Boxing.
Byte      bRef =  2;       // Constant in range: narrowing, then boxing.
// Byte  bRef2 =  257;     // Constant not in range. Compile-time error!

short s = 10;              // Narrowing from int to short.
// Integer   iRef1 = s;    // short not assignable to Integer.
Integer iRef3 = (int) s;   // Explicit widening with cast to int and boxing

boolean bv1 = boolRef;     // Unboxing.
byte  b1 = bRef;           // Unboxing.
int   iVal = bRef;         // Unboxing and widening.

Integer iRefVal = null;           // Always allowed.
// int j = iRefVal;               // NullPointerException at runtime.
if (iRef3 != null) iVal = iRef3;  // Avoids exception at runtime.
```

```
int tooBig   = Integer.MAX_VALUE + 1;    // -2147483648 which is Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;    //  2147483647 which is Integer.MAX_VALUE.
```

```
long notTooBig   = Integer.MAX_VALUE + 1L;   //  2147483648L in range.
long notTooSmall = Integer.MIN_VALUE - 1L;   // -2147483649L in range.
```

```
System.out.println( 4.0 / 0.0);          // Prints:  Infinity
System.out.println(-4.0 / 0.0);          // Prints: -Infinity
System.out.println(0.0 / 0.0);          // Prints: NaN
```

```
int value = - -10;                   // (-(-10)) is 10
int    sameSigns     = -4    * -8;   // result:  32
double oppositeSigns =  4    * -8.0; // Widening of int 4 to double. result: -32.0
int    zero          =  0    * -0;   // result:   0
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1.0; integer division, then widening conversion
double d2 = 4.0 / 8;       // result: 0.5
double d3 = 8 / 8.0;       // result: 1.0
float d4  = 12.0F / 8;     // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;       // ((12.0 / 4.0) * 3.0) which is 9.0
double result2 = 12.0 * 3.0 / 4.0;       // ((12.0 * 3.0) / 4.0) which is 9.0
```

```
int quotient  = 7 / 5;   // Integer division operation: 1
int remainder = 7 % 5;   // Integer remainder operation: 2
```

```
7 == (7 / 5) * 5 + (7 % 5)
  == (  1  ) * 5 + (7 % 5)
  ==             5 + (7 % 5)
2 ==                 (7 % 5)
```

(7 % 5) is equal to 2

```
7 == (7 / -5) * -5 + (7 % -5)
  == (  -1   ) * -5 + (7 % -5)
  ==             5 + (7 % -5)
2 ==                 (7 % -5)
```

(7 % –5) is equal to 2

```
-7 == (-7 / 5) * 5 + (-7 % 5)
   == (   -1   ) * 5 + (-7 % 5)
   ==             -5 + (-7 % 5)
-2 ==                  (-7 % 5)
```

(–7 % 5) is equal to –2

```
-7 == (-7 / -5) * -5 + (-7 % -5)
   == (   1   ) * -5 + (-7 % -5)
   ==              -5 + (-7 % -5)
-2 ==                   (-7 % -5)
```

(–7 % –5) is equal to –2

```
int  r0 =  7  %  7;     //  0
int  r1 =  7  %  5;     //  2
long r2 =  7L % -5L;    //  2L
int  r3 = -7  %  5;     // -2
long r4 = -7L % -5L;    // -2L
boolean relation = -7L == (-7L / -5L) * -5L + r4;  // true
```

```
double  dr0 =  7.0  %  7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
boolean fpRelation = dr2  == (-7.0) - (-5.0) * (long)(-7.0 / -5.0);  // true
float   fr3 = -7.0F %  0.0F;   // NaN
```

```
byte b = 3;            // int literal in range. Narrowing conversion.
b = (byte) -b;         // Cast required on assignment.
```

```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';                                 // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * -b) - (c / s);      // (1) 4888.0D
    System.out.println("Value of v: " + v);
  }
}
```

```
Byte   b = 10;       // Constant in range: narrowing and boxing on assignment.
Short  s = 20;       // Constant in range: narrowing and boxing on assignment.
```

```
char   c = 'z';       // 122 (\u007a)
int    i = s * b;     // Values in s and b promoted to int: unboxing, widening.
long   n = 20L + s;   // Value in s promoted to long: unboxing, widening.
float  r = s + c;     // Value in s is unboxed. This short value and the char
                      // value in c are promoted to int, followed by implicit
                      // widening conversion of int to float on assignment.
double d = r + i;     // Value in i promoted to float, followed by implicit
                      // widening conversion of float to double on assignment.
```

```
short h = 40;              // OK: int converted to short. Implicit narrowing.
h = h + 2;                 // Error: cannot assign an int to short.
```

```
h = (short) h + (short) 2;     // The resulting value should be cast.
```

```
h = (short) h + 2;              // The resulting value should be cast.
```

```
int i = 2;
i *= i + 4;             // (1) Evaluated as i = (int) ((i) * (i + 4)).

Integer iRef = 2;
iRef *= iRef + 4;      // (2) Evaluated as iRef = (Integer) ((iRef) * (iRef + 4)).

byte b = 2;
b += 10;                // (3) Evaluated as b = (byte) (b + 10).
b = b + 10;             // (4) Will not compile. Cast is required.
```

```
int[] a = new int[] { 2020, 2030, 2040 };
int i = 2;
a[i] += 1;      // Evaluates as a[2] = a[2] + 1, and a[2] gets the value 2041.
```

```
String strVal = "" + 2020;                      // (1) "2020"
String theName = " Uranium";
theName = " Pure" + theName;                    // (2) " Pure Uranium"
String trademark1 = 100 + "%" + theName;        // (3) "100% Pure Uranium"
String trademark2 = 100 + '%' + theName;      // (4) "137 Pure Uranium"
```

```
System.out.println("We can put two and two together and get " + 2 + 2);   // (5)
System.out.println("We can put two and two together and get " + (2 + 2)); // (6)
System.out.println("2 * 2 = " + 2 * 2);        // (7) 2 * 2 = 4
```

```
// (1) Prefix order: increment/decrement operand before use.
int i = 10;
int k = ++i + --i;  // ((++i) + (--i)). k gets the value 21 and i becomes 10.
--i;                // Only side effect utilized. i is 9. (expression statement)

Integer iRef = 11;  // Boxing on assignment
--iRef;             // Only side effect utilized. iRef refers to an Integer
                    // object with the value 10. (expression statement)
k = ++iRef + --iRef;// ((++iRef) + (--iRef)). k gets the value 21 and
                    // iRef refers to an Integer object with the value 10.

// (2) Postfix order: increment/decrement operand after use.
long j = 10;
long n = j++ + j--; // ((j++) + (j--)). n gets the value 21L and j becomes 10L.
j++;                // Only side effect utilized. j is 11L. (expression statement)
```

`k = ((++i) + (--i))`	Operands are evaluated from left to right.
`k = ( 11   + (--i))`	Side effect: `i += 1,` `i` gets the value 11.
`k = ( 11   +  10)`	Side effect: `i -= 1,` `i` gets the value 10.
`k = 21`	

```
byte b = 10;
int  i = ++b;          // i is 11, and so is b.
```

```
double x = 4.5;
x = x + ++x;           // x gets the value 10.0.
```

```
public class Assignment {
  public static void main(String[] args) {
    int a, b, c;
    b = 10;
```

```
      a = b = c = 20;
      System.out.println(a);
    }
  }
```

```
public class MyClass {
  public static void main(String[] args) {
    String a, b, c;
    c = new String("mouse");
    a = new String("cat");
    b = a;
    a = new String("dog");
    c = b;

    System.out.println(c);
  }
}
```

```
public class Prog1 {
  public static void main(String[] args) {
    int k = 1;
    int i = ++k + k++ + + k;     // (1)
    System.out.println(i);
  }
}
```

```
public class MyClass {
  public static void main(String[] args) {
    char c;
    int i;
    c = 'a'; // (1)
    i = c;   // (2)
    i++;     // (3)
    c = i;   // (4)
    c++;     // (5)
  }
}
```

```
public class EvaluationOrder {
  public static void main(String[] args) {
    int[] array = { 4, 8, 16 };
```

```
        int i = 1;
        array[++i] = --i;
        System.out.println(array[0] + array[1] + array[2]);
    }
}
```

```
double hours = 45.5;
Double time = 18.0;              // Boxing of double value.
boolean overtime = hours >= 35; // true. Binary numeric promotion: double <-- int.
boolean beforeMidnight = time < 24.0;// true. Unboxing of value in time reference.
char letterA = 'A';
boolean order = letterA < 'a';  // true. Binary numeric promotion: int <-- char.
```

```
int a = 1, b = 7, c = 10;
boolean illegal = a <= b <= c;       // (1) Illegal. Compile-time error!
boolean valid2 = a <= b && b <= c;   // (2) OK.
```

```
int year = 2002;
boolean isEven  = year % 2 == 0;     // true.
boolean compare = '1' == 1;          // false. Binary numeric promotion applied.
boolean test    = compare == false;  // true.
```

```
int a, b, c;
a = b = c = 5;
boolean illegal = a == b == c;                    // (1) Illegal.
boolean valid2 = a == b && b == c;                // (2) Legal.
boolean valid3 = a == b == true;                  // (3) Legal.
```

```
Pizza pizzaA = new Pizza("Sweet&Sour");      // new object
Pizza pizzaB = new Pizza("Sweet&Sour");      // new object
Pizza pizzaC = new Pizza("Hot&Spicy");       // new object

String banner = "Come and get it!";          // new object

boolean test  = banner == pizzaA;            // (1) Compile-time error
boolean test1 = pizzaA == pizzaB;            // false
boolean test2 = pizzaA == pizzaC;            // false

pizzaA = pizzaB;                        // Denote the same object; are aliases
boolean test3 = pizzaA == pizzaB;       // true
```

```
if (objRef != null) {
    // ... use objRef ...
}
```

```
Integer iRef = 10;
boolean b1 = iRef == null;         // Object reference equality
boolean b2 = iRef == 10;           // (1) Primitive data value equality
boolean b3 = null == 10;           // Compile-time error!
```

```
// Equality for String objects means identical character sequences.
String movie1 = new String("The Revenge of the Exception Handler");
String movie2 = new String("High Noon at the Java Corral");
String movie3 = new String("The Revenge of the Exception Handler");
boolean test0 = movie1.equals(movie2);             // false.
boolean test1 = movie1.equals(movie3);             // true.

// Equality for wrapper classes means same type and same primitive value.
Boolean flag1 = true;                              // Boxing.
Boolean flag2 = false;                             // Boxing.
boolean test2 = flag1.equals("true");              // false. Not same type.
boolean test3 = flag1.equals(!flag2);              // true. Same type and value.

Integer iRef = 100;                                // Boxing.
Short sRef = 100;                                  // Boxing <--- short <--- int
boolean test4 = iRef.equals(100);                  // true. Same type and value.
boolean test5 = iRef.equals(sRef);                 // false. Not same type.
boolean test6 = iRef.equals(3.14);                 // false. Not same type.
```

```
// The Pizza class does not override the equals() method, so we can use either
// equals() method inherited from the Object class or equality operator ==.
Pizza pizza1 = new Pizza("Veggies Delight");
Pizza pizza2 = new Pizza("Veggies Delight");
Pizza pizza3 = new Pizza("Cheese Delight");
boolean test7 = pizza1.equals(pizza2);             // false.
boolean test8 = pizza1.equals(pizza3);             // false.
boolean test9 = pizza1 == pizza2;                  // false.
pizza1 = pizza2;                                   // Creates aliases.
boolean test10 = pizza1.equals(pizza2);            // true.
boolean test11 = pizza1 == pizza2;                 // true.
```

```
if (i > 0 & i++ < 10) {/*...*/} // i will be incremented, regardless of value in i.
```

```
boolean b1, b2, b3 = false, b4 = false;
Boolean b5 = true;
b1 = 4 == 2 & 1 < 4;           // false, evaluated as (b1 = ((4 == 2) & (1 < 4)))
b2 = b1 | !(2.5 >= 8);         // true
b3 = b3 ^ b5;                  // true, unboxing conversion on b5
b4 = b4 | b1 & b2;             // false
```

```
   (b4 = (b4 | (b1 & b2)))
⇒ (b4 = (false | (b1 & b2)))
⇒ (b4 = (false | (false & b2)))
⇒ (b4 = (false | (false & true)))
⇒ (b4 = (false | false))
⇒ (b4 = false)
⇒ false
```

```
boolean b1 = false, b2 = true, b3 = false;
Boolean b4 = false;
b1 |= true;             // true
b4 ^= b1;               // (1) true, unboxing in (b4 ^ (b1)), boxing on assignment
b3 &= b1 | b2;          // (2) false, b3 = (b3 & (b1 | b2))
b3 = b3 & b1 | b2;      // (3) true,  b3 = ((b3 & b1) | b2)
```

```
Boolean b1 = 4 == 2 && 1 < 4;   // false, short-circuit evaluated as
                                // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;    // true, short-circuit evaluated as
                                // (b2 = ((!b1) || (2.5 > 8)))
Boolean b3 = !(b1 && b2);       // true
boolean b4 = b1 || !b3 && b2;   // false, short-circuit evaluated as
                                // (b4 = (b1 || ((!b3) && b2)))
```

```
   (b4 = (b1 || ((!b3) && b2)))
⟹ (b4 = (false || ((!b3) && b2)))
⟹ (b4 = (false || ((!true) && b2)))
⟹ (b4 = (false || ((false) && b2)))
⟹ (b4 = (false || false))
⟹ (b4 = false)
```

```
public class ShortCircuit {
  public static void main(String[] args) {
    // Boolean b1 = 4 == 2 && 1 < 4;
    Boolean b1 = operandEval(1, 4 == 2) && operandEval(2, 1 < 4);
    System.out.println("Value of b1: " + b1);

    // boolean b2 = !b1 || 2.5 > 8;
    boolean b2 = !operandEval(1, b1) || operandEval(2, 2.5 > 8);
    System.out.println("Value of b2: " + b2);

    // Boolean b3 = !(b1 && b2);
    Boolean b3 = !(operandEval(1, b1) && operandEval(2, b2));
    System.out.println("Value of b3: " + b3);

    // boolean b4 = b1 || !b3 && b2;
    boolean b4 = operandEval(1, b1) || !operandEval(2, b3) && operandEval(3, b2);
    System.out.println("Value of b4: " + b4);

    // boolean b5 = b1 | !b3 & b2;    // Using boolean logical operators
    boolean b5 = operandEval(1, b1) | !operandEval(2, b3) & operandEval(3, b2);
    System.out.println("Value of b5: " + b5);
  }
  static boolean operandEval(int opNum, boolean operand) {                // (1)
    System.out.println(opNum);
    return operand;
  }
}
```

```
1
Value of b1: false
1
Value of b2: true
1
Value of b3: true
1
2
Value of b4: false
1
2
3
Value of b5: false
```

```
if (objRef != null && objRef.equals(other)) { /*...*/ }
if (i > 0 && i++ < 10) {/*...*/}   // i is not incremented if i > 0 is false.
if (i > 0 || i++ < 10) {/*...*/}   // i is not incremented if i > 0 is true.
```

```
char v1 = ')';          // Unicode value 41
byte v2 = 13;

int result1 = ~v1;      // -42
int result2 = v1 & v2;  // 9
int result3 = v1 | v2;  // 45
int result4 = v1 ^ v2;  // 36
```

```
public class BitOperations {
  public static void main(String[] args) {
    char v1 = ')';                      // Unicode value 41
    byte v2 = 13;
    printIntToStr("v1:", v1);           // 41
    printIntToStr("v2:", v2);           // 13
    printIntToStr("~v1:", ~v1);         // -42
    printIntToStr("v1 & v2:", v1 & v2); // 9
    printIntToStr("v1 | v2:", v1 | v2); // 45
    printIntToStr("v1 ^ v2:", v1 ^ v2); // 36
  }

  public static void printIntToStr(String label, int result) {
    System.out.println(label);
    System.out.println("    Binary:  " + Integer.toBinaryString(result));
    System.out.println("    Hex:     " + Integer.toHexString(result));
    System.out.println("    Decimal: " + result);
  }
}
```

```
v1:
    Binary:  101001
    Hex:     29
    Decimal: 41
v2:
    Binary:  1101
    Hex:     d
    Decimal: 13
~v1:
    Binary:  11111111111111111111111111010110
    Hex:     ffffffd6
    Decimal: -42
v1 & v2:
    Binary:  1001
    Hex:     9
    Decimal: 9
v1 | v2:
    Binary:  101101
    Hex:     2d
    Decimal: 45
v1 ^ v2:
    Binary:  100100
    Hex:     24
    Decimal: 36
```

```
int  v0 = -42;
char v1 = ')';  // 41
byte v2 = 13;

v0 &= 15;       //     1...1101 0110 & 0...0000 1111 => 0...0000 0110 (= 6)
v1 |= v2;       // (1) 0...0010 1001 | 0...0000 1101 => 0...0010 1101 (= 45, '-')
```

```
int i = 12;
int result = i << 4;    // 192
```

```
i << 4
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

```
byte  b = -42;              // 11010110
short n = 4;
int   result = b << n;    // -672
```

```
b << n
= 1101 0110 << 0000 0000 0000 0100
= 1111 1111 1111 1111 1111 1111 1101 0110 <<0000 0000 0000 0000 0000 0000 0000 0100
= 1111 1111 1111 1111 1111 1111 1101 0110 << 4
= 1111 1111 1111 1111 1111 1101 0110 0000
= 0xfffffd60
= -672
```

```
byte a = 32;
int  j = a << 3;              // 256
byte b = (byte) (a << 3);     // 0. Cast mandatory.
```

```
a << 3
= 0000 0000 0000 0000 0000 0000 0010 0000 << 3
= 0000 0000 0000 0000 0000 0001 0000 0000
= 0x00000100
= 256
```

```
12 << 36
= 0000 0000 0000 0000 0000 0000 0000 1100 << (0...0010 0100 & 0001 1111)
= 0000 0000 0000 0000 0000 0000 0000 1100 << 0...0000 0100
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

```
int i = 12;
int result = i >> 2;     // 3
```

```
i >> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

```
byte b = -42;          // 11010110
int result = b >> 4;   // -3
```

```
b >> 4
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 4
= 1111 1111 1111 1111 1111 1111 1111 1101
= 0xfffffffd
= -3
```

```
-42 >> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >> (1...1111 1100 & 0001 1111)
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 0...0001 1100
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 28
= 1111 1111 1111 1111 1111 1111 1111 1111
= 0xffffffff
= -1
```

```
12 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

```
byte b = -42;                  // 1101 0110
int result = b >>> 4;          // 268435453
```

```
b >>> 4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 4
= 0000 1111 1111 1111 1111 1111 1111 1101
= 0x0ffffffd
= 268435453
```

```
-42 >>> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 28
= 0000 0000 0000 0000 0000 0000 0000 1111
= 0x0000000f
= 15
```

```
int i = -42;
i >>= 4;                        // 1...1101 0110 >> 4 => 1...1111 1101 (= -3).

byte a = 12;
a <<= 5;                        // (1) -128. Evaluated as a = (byte)((int)a << 5)
a = a << 5;                     // Compile-time error! Needs explicit cast.
```

```
a = (byte) (a << 5)
  = (byte) (0000 0000 0000 0000 0000 0000 0000 1100 << 5)
  = (byte)  0000 0000 0000 0000 0000 0001 1000 0000
  = 1000 0000
  = 0x80
  = -128
```

```
boolean leapYear = false;
byte v29 = 29;
byte v28 = 28;
int daysInFebruary = leapYear ? v29 : v28;   // (1)
int i = 3;
int j = 4;
int minValue1 = i < j ? i : j;                       // (2) int
int minValue2 = i < j ? i : Double.MIN_VALUE;        // (3) double. Not OK.
double minDoubleValue = i < j ? i : Double.MIN_VALUE; // (4) double
```

```
// Assume i and j are of type int and initialized correctly.
Object obj = i < j ? i : true;        // (5) value of i boxed in Integer or
                                      //     literal true boxed in Boolean
System.out.println(i < j ? i : "Hi"); // (6) value of i boxed in Integer or
                                      //     String object "Hi"
```

```
(i < j) ? i : j;    // Compile-time error!
```

`a?b:c?d:e?f:g` evaluates as `(a?b:(c?d:(e?f:g)))`

```
int n = 3;
String msg = (n==0) ? "no cookies." : (n==1) ? "one cookie." : "many cookies.";
System.out.println("You get " + msg); // You get many cookies.
```

```
Pizza onePizza = new Pizza();       // Create an instance of the Pizza class.
```

```
int[] anArray = new int[5];// Declare and construct an int array of 5 elements.
anArray[4] = anArray[3];   // Element at index 4 gets value of element at index 3.
```

```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza; // true.
boolean test2 = "Pizza" instanceof Pizza; // Compile-time error. Incompatible
                                          // operand types.
boolean test3 = null instanceof Pizza; // Always false. null is not an instance.
```

```
java.util.function.Predicate<String> predicate = str -> str.length() % 2 == 0;
boolean test4 = predicate.test("The lambda strikes back!");    // true.
```

```
public class BoolOp {
  static void op(boolean a, boolean b) {
    boolean c = a != b;
    boolean d = a ^ b;
    boolean e = c == d;
    System.out.println(e);
  }

  public static void main(String[] args) {
    op(false, false);
    op(true, false);
    op(false, true);
    op(true, true);
  }
}
```

```
public class OperandOrder {
  public static void main(String[] args) {
    int i = 0;
    int[] a = {3, 6};
    a[i] = i = 9;
    System.out.println(i + " " + a[0] + " " + a[1]);
  }
}
```

```
public class Logic {
  public static void main(String[] args) {
    int i = 0;
    int j = 0;

    boolean t = true;
    boolean r;

    r = (t &  0 < (i+=1));
    r = (t && 0 < (i+=2));
    r = (t |  0 < (j+=1));
    r = (t || 0 < (j+=2));
    System.out.println(i + " " + j);
  }
}
```

```
int x = 1, y = 2, z = 3;
if (x < y || ++z > 4) {
  System.out.println("a" + x + y + z);
}
if (x < y && ++z > 4) {
  System.out.println("b" + x + y + z);
}
```

```
public class RQ05A200 {
  public static void main(String[] args) {
    int i = 20;
    int j = 30;
    // (1) INSERT STATEMENT HERE
  }
}
```

```
public class RQ05A100 {
  public static void main(String[] args) {
    int n1 = 10, n2 = 10;
    int m1 = 20, m2 = 30;
    int result = n1 != n2? n1 : m1 != m2? m1 : m2;
    System.out.println(result);
  }
}
```

```
class_modifiers class class_name extends_clause implements_clause // Class header
{ // Class body
    field_declarations
    method_declarations
    constructor_declarations
    member_type_declarations
}
```

```
method_modifiers return_type method_name
                  (formal_parameter_list)  throws_clause // Method header
{ // Method body
    local_variable_declarations
    statements
}
```

```
void setSpeed(var newSpeed) {}          // var not permitted. Compile-time error!
```

```
double cubeVolume(double length, double width, double height) {}
```

`cubeVolume(double, double, double)`

```
char a, b, c;              // a, b, and c are character variables.
double area;               // area is a floating-point variable.
boolean flag;              // flag is a boolean variable.
```

```
int i = 10,              // i is an int variable with initial value 10.
    j = 0b101;           // j is an int variable with initial value 5.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class Hamburger,
          smallOne; // and so can variable smallOne.
```

```
Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration statement
```

```
public class Light {
  // Static variable
  static int counter;      // Default value 0 when class is loaded

  // Instance variables:
  int     noOfWatts = 100; // Explicitly set to 100
  boolean indicator;       // Implicitly set to default value false
  String  location;        // Implicitly set to default value null

  public static void main(String[] args) {
    Light bulb = new Light();
    System.out.println("Static variable counter:     " + Light.counter);
    System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
    System.out.println("Instance variable indicator: " + bulb.indicator);
    System.out.println("Instance variable location:  " + bulb.location);
  }
}
```

```
Static variable counter:     0
Instance variable noOfWatts: 100
Instance variable indicator: false
Instance variable location:  null
```

```
public class TooSmartClass {
  public static void main(String[] args) {
    double weight = 10.0, thePrice;                    // (1) Local variables

    if (weight <  10.0) thePrice = 20.50;
    if (weight >  50.0) thePrice = 399.00;
    if (weight >= 10.0) thePrice = weight * 10.0;      // (2) Always executed
    System.out.println("The price is: " + thePrice);   // (3) Compile-time error!
  }
}
```

```
double weight = 10.0, thePrice = 0.0;    // (1') Both local variables initialized
```

```
public class VerySmartClass {
  public static void main(String[] args) {
    String importantMessage;       // Local reference variable

    System.out.println("The message length is: " +
                       importantMessage.length());  // Compile-time error!
  }
}
```

```
String importantMessage = "Initialize before use!";
```

```
public class Light {
  // Fields:
  int     noOfWatts;      // Wattage
  boolean indicator;      // On or off
  String  location;       // Placement

  // Constructor:
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;

    this.noOfWatts = noOfWatts;   // (1) Assignment to field
    indicator = indicator;        // (2) Assignment to parameter
    location = site;              // (3) Assignment to local variable
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }

  // Instance method:
  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  // Static method:
  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

```
Current object: Light@1bc4459
Current object: Light@1bc4459
No. of watts: 100
Indicator:    false
Location:     null
```

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

`methodA(int, double)`	1'
`methodA(int)j`	2': Number of parameters
`methodA()`	3': Number of parameters
`methodA(double, int)`	4': Order of parameters
`methodA(int, double)`	5': Same as 1'

```
void bake(Cake k)  { /* ... */ }                    // (1)
void bake(Pizza p) { /* ... */ }                    // (2)

int     halfIt(int a) { return a/2; }               // (3)
double  halfIt(int a) { return a/2.0; }             // (4) Not OK. Same signature.
```

```
access_modifier class_name (formal_parameter_list)
                                    throws_clause // Constructor header
{ // Constructor body
    local_variable_declarations
    statements
}
```

```
public class Name {

  Name() {                        // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                   // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();            // (3) Constructor call followed by method call
  }
}
```

```
class_name() { super(); }   // No parameters. Calls superclass constructor.
```

```
class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // No constructors
  //...
}

class Greenhouse {
  // ...
  Light oneLight = new Light();     // (1) Call to default constructor
}
```

```
class Light {
  // ...
  Light() {                        // (1) No-argument constructor
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  //...
}

class Greenhouse {
  // ...
  Light extraLight = new Light();   // (2) Call to no-argument constructor
}
```

```
class Light {
  // ...
  // Only non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {          // (1)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}
```

```
class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");// (2) OK
  Light firstLight = new Light();                       // (3) Compile-time error
}
```

```
class Light {
  // ...
  // No-argument constructor:
  Light() {                                                     // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light firstLight = new Light();                          // (3) OK. Calls (1)
  Light moreLight  = new Light(100, true, "Greenhouse"); // (4) OK. Calls (2)
}
```

```
// File: StaticTest.java
import static java.lang.System.out;

class Light {

  // Static field:
  static int counter;                  // (1) No initializer expression

  // Static method:
  public static void printStatic() {
    Light myLight = null;
    out.printf("%s, %s, %s%n", counter, Light.counter, myLight.counter); // (2)
    long counter = 10;                 // (3) Local variable shadows static field
    out.println("Local counter: " + counter);       // (4) Local variable accessed
    out.println("Static counter: " + Light.counter);// (5) Static field accessed

//  out.println(this.counter);         // (6) Cannot use this in static context
//  printNonStatic();                  // (7) Cannot call non-static method
  }

  // Non-static method:
  public void printNonStatic() {
   out.printf("%s, %s, %s%n", counter, this.counter, Light.counter);     // (8)
  }
}
//________________________________________________________________________________
public class StaticTest {              // Client of class Light
  public static void main(String[] args) {
    Light.counter++;                   // (9) Using class name
    Light dimLight = null;
    dimLight.counter++;                // (10) Using object reference

    out.print("Light.counter == dimLight.counter: ");
    out.println(Light.counter == dimLight.counter);//(11) Aliases for static field

    out.println("Calling static method using class name:");
    Light.printStatic();               // (12) Using class name
    out.println("Calling static method using object reference:");
    dimLight.printStatic();            // (13) Using object reference
  }
}
```

```
Light.counter == dimLight.counter: true
Calling static method using class name:
2, 2, 2
Local counter: 10
Static counter: 2
Calling static method using object reference:
2, 2, 2
Local counter: 10
Static counter: 2
```

```
public class Init {

  String title;
  boolean published;

  static int total;
  static double maxPrice;

  public static void main(String[] args) {
    Init initMe = new Init();
    double price;
    if (true)
      price = 100.00;
    System.out.println("|" + initMe.title + "|" + initMe.published + "|" +
                       Init.total + "|" + Init.maxPrice + "|" + price + "|");
  }
}
```

```
void fly(int distance) {}
int  fly(int time, int speed) { return time*speed; }

void fall(int time) {}
int  fall(int distance) { return distance; }

void glide(int time) {}
void Glide(int time) {}
```

```
public class MyClass {
  long var;

  public void MyClass(long param) { var = param; }  // (1)

  public static void main(String[] args) {
    MyClass a, b;
    a = new MyClass();                                // (2)
    b = new MyClass(5);                               // (3)
  }
}
```

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

array_name = `new` *element_type*[*array_size*];

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

```
anIntArray   = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
largePizzas  = new Pizza[3];         // array of 3 pizzas
```

element_type$_1$`[]` *array_name* `= new` *element_type*$_2$`[`*array_size*`];`

```
int[] anIntArray = new int[10];                  // Default element value: 0
Pizza[] mediumPizzas = new Pizza[5];             // Default element value: null
```

array_name = `new` *element_type*`[`*array_size*`];`

```
int[] anIntArray = {13, 49, 267, 15, 215};
```

```
Pizza[] pizzaOrder = { new Pizza(), new Pizza(), null };
```

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

```
// Array with 4 String objects:
String[] pets = {"crocodiles", "elephants", "crocophants", "elediles"}; // (1)

// Array of 3 characters:
char[] charArray = {'a', 'h', 'a'};    // (2) Not the same as "aha"
```

array_name [*index_expression*]

```
char letter = "AHA".toCharArray()[1];     // 'H'
```

```
public class Trials {
  public static void main(String[] args) {
    // Declare and construct the local arrays:
    double[] storeMinimum = new double[5];               // (1)
    double[] trialArray = new double[15];                // (2)
    for (int i = 0; i < storeMinimum.length; ++i) {      // (3)
      // Initialize the array.
      randomize(trialArray);

      // Find and store the minimum value.
      storeMinimum[i] = findMinimum(trialArray);
    }
    // Print the minimum values:                                    (4)
    for (double minValue : storeMinimum)
      System.out.printf("%.4f%n", minValue);
  }

  public static void randomize(double[] valArray) {            // (5)
    for (int i = 0; i < valArray.length; ++i)
      valArray[i] = Math.random() * 100.0;
  }

  public static double findMinimum(double[] valArray) {  // (6)
    // Assume the array has at least one element.
    double minValue = valArray[0];
    for (int i = 1; i < valArray.length; ++i)
      minValue = Math.min(minValue, valArray[i]);
    return minValue;
  }
}
```

element_type$_1$[] *array_name* = `new` *element_type*$_2$[*array_size*];   // (1)

```
int[] intArray = new int[5];
```

```
element_type[] array_name = { array_initialize_list };                    // (2)
int[] intArray = {3, 5, 2, 8, 6};
```

`new` *element_type*`[]` { *array_initialize_list* }

```
new int[] {3, 5, 2, 8, 6}
```

```
int[] intArray = {3, 5, 2, 8, 6};                                 // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6};                       // (2)
```

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31};                   // Compile-time error
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // OK
```

```
public class AnonArray {
  public static void main(String[] args) {
    System.out.println("Minimum value: " +
        findMinimum(new int[] {3, 5, 2, 8, 6}));                   // (1)
  }

  public static int findMinimum(int[] dataSeq) {                   // (2)
    // Assume the array has at least one element.
    int min = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index] < min)
        min = dataSeq[index];
    return min;
  }
}
```

element_type`[][]...[]` *array_name*`;`

element_type array_name`[][]...[];`

```
int[][] mXnArray;       // two-dimensional array
int[]   mXnArray[];     // two-dimensional array
int     mXnArray[][];  // two-dimensional array
```

```
int[][] mXnArray = new int[4][5];    // 4 x 5 matrix of ints
```

```
double[][] identityMatrix = {
  {1.0, 0.0, 0.0, 0.0 }, // 1. row
  {0.0, 1.0, 0.0, 0.0 }, // 2. row
  {0.0, 0.0, 1.0, 0.0 }, // 3. row
  {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 floating-point matrix
```

```
Pizza[][] pizzaGalore = {
  { new Pizza(), null, new Pizza() },    // 1. row is an array of 3 elements.
  { null, new Pizza()},                  // 2. row is an array of 2 elements.
  new Pizza[1],                          // 3. row is an array of 1 element.
  {},                                    // 4. row is an array of 0 elements.
  null                                   // 5. row is not constructed.
};
```

```
HotelRoom[][][][] rooms = new HotelRoom[10][5][][];  // Just streets and hotels.
```

```
rooms[0][0]       = new HotelRoom[3][]; // 3 floors in 1st hotel on 1st street.
rooms[0][0][0]    = new HotelRoom[8];   // 8 rooms on 1st floor in this hotel.
rooms[0][0][0][0] = new HotelRoom();    // Initializes 1st room on this floor.
```

```
double[][] matrix = new double[3][];      // (1) Number of rows.

for (int i = 0; i < matrix.length; ++i)
  matrix[i] = new double[i + 1];          // Construct a row.
```

```
double[][] matrix2 = {    // (2) Using array initializers.
  {1.0},                  // 1. row
  {1.0, 2.0},             // 2. row
  {1.0, 2.0, 3.0}         // 3. row
};

double[][] matrix3 = new double[][] { // (3) Using an anonymous array of arrays.
  {1.0},                  // 1. row
  {1.0, 2.0},             // 2. row
  {1.0, 2.0, 1.0}         // 3. row
};
```

```
public class MultiArrays {

  public static void main(String[] args) {
    // Declare and construct the M X N matrix.
    int[][] mXnArray = {                                                 // (1)
```

```
        {16,  7, 12}, // 1. row
        { 9, 20, 18}, // 2. row
        {14, 11,  5}, // 3. row
        { 8,  5, 10}  // 4. row
    }; // 4 x 3 int matrix

    // Find the minimum value in an M X N matrix:
    int min = mXnArray[0][0];
    for (int i = 0; i < mXnArray.length; ++i)                      // (2)
      // Find min in mXnArray[i], in the row given by index i:
      for (int j = 0; j < mXnArray[i].length; ++j)                 // (3)
        min = Math.min(min, mXnArray[i][j]);

    System.out.println("Minimum value: " + min);
  }
}''
```

object_reference.*method_name*(*actual_parameter_list*)

class_name.*static_method_name*(*actual_parameter_list*)

method_name(*actual_parameter_list*)

```
objRef.doIt(time, place);         // Explicit object reference
int i = java.lang.Math.abs(-1);   // Fully qualified class name
int j = Math.abs(-1);             // Simple class name
someMethod(ofValue);              // Object or class is implied
someObjRef.make().make().make();  // make() returns a reference value
```

```
static void doIt(long i) { /* ... */ }
```

```
Integer intRef = 34;
Long longRef = 34L;
doIt(34);         // (1) Primitive widening conversion: long <-- int
doIt(longRef);    // (2) Unboxing: long <-- Long
doIt(intRef);     // (3) Unboxing, followed by primitive widening conversion:
                  //     long <-- int <-- Integer
```

```
public class CustomerOne {
  public static void main (String[] args) {
    PizzaFactory pizzaHouse = new PizzaFactory();
    int pricePrPizza = 15;
    System.out.println("Value of pricePrPizza before call: " + pricePrPizza);
    double totPrice = pizzaHouse.calcPrice(4, pricePrPizza);              // (1)
    System.out.println("Value of pricePrPizza after call: " + pricePrPizza);
  }
}
```

```
class PizzaFactory {
  public double calcPrice(int numberOfPizzas, double pizzaPrice) {       // (2)
    pizzaPrice = pizzaPrice / 2.0;       // Changes price.
    System.out.println("Changed pizza price in the method: " + pizzaPrice);
    return numberOfPizzas * pizzaPrice;
  }
}
```

```
Value of pricePrPizza before call: 15
Changed pizza price in the method: 7.5
Value of pricePrPizza after call: 15
```

```
public class CustomerTwo {
  public static void main (String[] args) {
    Pizza favoritePizza = new Pizza();              // (1)
    System.out.println("Meat on pizza before baking: " + favoritePizza.meat);
    bake(favoritePizza);                            // (2)
    System.out.println("Meat on pizza after baking: " + favoritePizza.meat);
  }
  public static void bake(Pizza pizzaToBeBaked) {   // (3)
    pizzaToBeBaked.meat = "chicken";  // Change the meat on the pizza.
    pizzaToBeBaked = null;                          // (4)
  }
}

class Pizza {                                       // (5)
  String meat = "beef";
}
```

Meat on pizza before baking: beef
Meat on pizza after baking: chicken

```
public class Percolate {

  public static void main (String[] args) {
    int[] dataSeq = {8,4,6,2,1};    // Create and initialize an array.

    // Write array before percolation:
    printIntArray(dataSeq);

    // Percolate:
    for (int index = 1; index < dataSeq.length; ++index)
```

```
      if (dataSeq[index-1] > dataSeq[index])
        swap(dataSeq, index-1, index);                    // (1)

    // Write array after percolation:
    printIntArray(dataSeq);
  }

  public static void swap(int[] intArray, int i, int j) { // (2)
    int tmp = intArray[i]; intArray[i] = intArray[j]; intArray[j] = tmp;
  }

  public static void swap(int v1, int v2) {               // (3) Logical error!
    int tmp = v1; v1 = v2; v2 = tmp;
  }

  public static void printIntArray(int[] array) {         // (4)
    for (int value : array)
      System.out.print(" " + value);
    System.out.println();
  }
}
```

```
swap(dataSeq[index-1], dataSeq[index]); // Call signature: swap(int, int)
```

```
public class FindMinimum {

  public static void main(String[] args) {
    int[] dataSeq = {6,4,8,2,1};

    int minValue = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      minValue = minimum(minValue, dataSeq[index]);           // (1)

    System.out.println("Minimum value: " + minValue);
  }

  public static int minimum(int i, int j) {                   // (2)
    return (i <= j) ? i : j;
  }
}
```

```
public class FindMinimumMxN {

  public static void main(String[] args) {
    int[][] matrix = { {8,4},{6,3,2},{7} };                  // (1)

    int min = findMinimum(matrix[0]);                        // (2)
    for (int i = 1; i < matrix.length; ++i) {
      int minInRow = findMinimum(matrix[i]);                 // (3)
      min = Math.min(min, minInRow);
    }
    System.out.println("Minimum value in matrix: " + min);
  }

  public static int findMinimum(int[] seq) {                 // (4)
    int min = seq[0];
    for (int i = 1; i < seq.length; ++i)
      min = Math.min(min, seq[i]);
    return min;
  }
}
```

```
public double calcPrice(int numberOfPizzas, final double pizzaPrice) {  // (2')
  pizzaPrice = pizzaPrice/2.0;         // (3) Not allowed. Compile-time error!
  return numberOfPizzas * pizzaPrice;
}
```

```
public static void bake(final Pizza pizzaToBeBaked) { // (3)
  pizzaToBeBaked.meat = "chicken";   // (3a) Allowed
  pizzaToBeBaked = null;             // (4) Not allowed. Compile-time error!
}
```

type... *formal_parameter_name*

```
public static void publish(int n, String... data) {      // (int, String[])
  System.out.println("n: " + n + ", data size: " + data.length);
}
```

```
publish(1);                     // (1, new String[] {})
publish(2, "two");              // (2, new String[] {"two"})
publish(3, "two", "three");  // (3, new String[] {"two", "three"})
```

```
public static void transmit(String... data) {  }  // Compile-time error!
public static void transmit(String[] data)  {  }  // Compile-time error!
public static void flexiPrint(Object... data) { // Object[]
  //...
}
```

```
public class VarargsDemo {
  public static void flexiPrint(Object... data) { // Object[]
    // Print the name of the Class object for the varargs parameter.
    System.out.print("Type: " + data.getClass().getName());

    System.out.println("  No. of elements: " + data.length);

    System.out.print("Element values: ");
    for(Object element : data)
      System.out.print(element + " ");
    System.out.println();
  }

  public static void main(String... args) {
    int    day       = 13;
    String monthName = "August";
    int    year      = 2009;

    // Passing primitives and non-array types:
    flexiPrint();                      // (1) new Object[] {}
    flexiPrint(day);                   // (2) new Object[] {Integer.valueOf(day)}
    flexiPrint(day, monthName);        // (3) new Object[] {Integer.valueOf(day),
                                       //                   monthName}
    flexiPrint(day, monthName, year);  // (4) new Object[] {Integer.valueOf(day),
                                       //                   monthName,
                                       //                   Integer.valueOf(year)}
    System.out.println();

    // Passing an array type:
    Object[] dateInfo = {day,          // (5) new Object[] {Integer.valueOf(day),
                         monthName,    //                   monthName,
                         year};        //                   Integer.valueOf(year)}
    flexiPrint(dateInfo);              // (6) Non-varargs call
    flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
    flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
    System.out.println();
    // Explicit varargs or non-varargs call:
    flexiPrint(args);                  // (9) Warning!
    flexiPrint((Object) args);         // (10) Explicit varargs call
    flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
  }
}
```

```
>javac VarargsDemo.java
VarargsDemo.java:41: warning: non-varargs call of varargs method with inexact
argument type for last parameter;
    flexiPrint(args);                  // (9) Warning!
               ^
  cast to Object for a varargs call
  cast to Object[] for a non-varargs call and to suppress this warning
1 warning
```

```
>java VarargsDemo To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 0                    (1)
Element values:
Type: [Ljava.lang.Object;  No. of elements: 1                    (2)
Element values: 13
Type: [Ljava.lang.Object;  No. of elements: 2                    (3)
Element values: 13 August
Type: [Ljava.lang.Object;  No. of elements: 3                    (4)
Element values: 13 August 2009

Type: [Ljava.lang.Object;  No. of elements: 3                    (6)
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 1                    (7)
Element values: [Ljava.lang.Object;@1eed786
Type: [Ljava.lang.Object;  No. of elements: 1                    (8)
Element values: [Ljava.lang.Object;@1eed786

Type: [Ljava.lang.String;  No. of elements: 6                    (9)
Element values: To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 1                    (10)
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6                    (11)
Element values: To arg or not to arg
```

```
    flexiPrint(dateInfo);                  // (6) Non-varargs call
```

```
flexiPrint((Object) dateInfo);      // (7) new Object[] {(Object) dateInfo}
flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
```

```
    flexiPrint(args);                       // (9) Warning!
    flexiPrint((Object) args);          // (10) Explicit varargs call
    flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
```

```
public static void main(String[] args)    // Method header
public static void main(String args[])    // Method header
public static void main(String... args)   // Method header
```

```
>java Colors red yellow green "blue velvet"
```

```
public class Colors {
   synchronized public static void main(String[] args) {
    System.out.println("No. of program arguments: " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println("Argument no. " + i + " (" + args[i] + ") has " +
                         args[i].length() + " characters.");
  }
}
```

```
>java Colors red yellow green "blue velvet"
No. of program arguments: 4
Argument no. 0 (red) has 3 characters.
Argument no. 1 (yellow) has 6 characters.
Argument no. 2 (green) has 5 characters.
Argument no. 3 (blue velvet) has 11 characters.
```

```
// Class ValidLVTI illustrates valid use of the restricted type name var.
public class ValidLVTI {

  // Static initializer block:
  static {
    var slogan = "Keep calm and code Java.";        // (1a) Allowed in static
  }                                                 //      initializer block
```

```
// Instance initializer block:
{
  var banner = "Keep calm and catch exceptions."; // (1b) Allowed in instance
}                                                 //      initializer block

// Constructor:
public ValidLVTI() {
  var luckyNumber = 13;                         // (2) Allowed in a constructor.
}

// Method:
public static void main(String[] args) {

  var virus = "COVID-19";                       // (3a) Type of virus is String.
  var acronym = virus.substring(0, 5);          // (3b) Type of acronym is String.
  var num = Integer.parseInt(virus.substring(6)); // (3c) Type of num is int.

  var obj = new Object();                       // (4) Type of obj is Object.
  var title = (String) null; // (5) Initialization expression type is String.
                             //     Type of title is String.
  var sqrtOfNumber = Math.sqrt(100); // (6) Type of sqrtOfNumber is double,
                                     //     since the method returns
                                     //     a double value.

  var tvSize  = (short) 55;  // (7) Type of tvSize is short.
  var tvSize2 = 65;          // (8) Type of tvSize2 is int.

  var diameter = 10.0;       // (9) Type of diameter is double.
  var radius = 2.5F;         // (10) Type of radius is float.

  // Arrays:
  var vowels = new char[] {'a', 'e', 'i', 'o', 'u' }; // (11a) Type of vowels
                                                      // is char[]. Size is 5.
  var zodiacSigns = new String[12]; // (11b) Type of zodiacSigns is String[].
                                    //       Size is 12.
  var a_2x3 = new int[2][3]; // (11c) Type of a_2x3 is int[][]. Size is 2x3.
  var a_2xn = new int[2][];  // (11d) Type of a_2xn is int[][]. Size is 2x?,
                             //       where second dimension can be undefined.

  // The for(:) loop:
  var word1 = "";            // (12) Type of word2 is String.
  for (var vowel : vowels) { // (13) Type of vowel is char in the for(:)loop.
    var letter = vowel;      // (14) Type of letter is char.
    word1 += letter;
  }

  // The for(;;) loop:
  var word2 = "";                              // (15) Type of word2 is String.
  for (var i = 0; i < vowels.length; i++) { // (16) Type of i is int in
                                               //      the for loop.
    var letter = vowels[i];                    // (17) Type of letter is char.
    word2 += letter;
  }
```

```
    // switch-statement:
    switch(virus) {
      case "Covid-19":
        var flag = "Needs to be tested.";      // (18) Type is String.
        // Do testing.
        break;
      default: // Do nothing.
    }
  }
}
```

```
// Class InvalidLVTI illustrates invalid use of the restricted type name var.
public class InvalidLVTI {

  var javaVendor = "Oracle"; // (19) Not allowed in instance variable declaration.

  static var javaVersion = 11; // (20) Not allowed in static variable declaration.

  public static void main(var args) { // (21) Not allowed for method parameters.

    var name;              // (22) Not allowed without initialization expression.

    var objRef = null;     // (23) Literal null not allowed.

    var x = 10.0, y = 20.0, z = 40;   // (24) Not allowed in compound declaration.

    var vowelsOnly = {'a', 'e', 'i', 'o', 'u' }; // (25) Array initializer not
                                                  //      allowed.
    var attendance = new int[];          // (26) Non-empty dimension required.
    var array3Dim = new String[][2][];  // (27) Cannot specify an empty dimension
                                         //      before a non-empty dimension.
    var letters[] = new char[]{'a', 'e', 'i', 'o', 'u' }; // (28) var not allowed
                                                           //      as element type.

    var prompt = prompt + 1;             // (29) Self-reference not allowed in
                                         //      initialization expression.
  }

  public static var getPlatformName() { // (30) Not allowed as return type.
    return "JDK";
  }
}
```

```
public class var {}   // var is not permitted as a class name. Compile-time error!
```

```
int []aa[] = new int [4][4];
int bb[][] = new int [4][4];
int cc[][] = new int [][4];
int []dd[] = new int [4][];
int [][]ee = new int [4][4];
```

```
public class MyClass {
  public static void main(String[] args) {
    int size = 20;
    int[] arr = new int[ size ];
    for (int i = 0; i < size; ++i) {
      System.out.println(arr[i]);
    }
  }
}
```

```java
public class DefaultValuesTest {
  int[] ia = new int[1];
  boolean b;
  int i;
  Object o;

  public static void main(String[] args) {
    DefaultValuesTest instance = new DefaultValuesTest();
    instance.print();
  }

  public void print() {
    System.out.println(ia[0] + " " + b + " " + i + " " + o);
  }
}
```

```
public class ParameterPass {
  public static void main(String[] args) {
    int i = 0;
    addTwo(i++);
    System.out.println(i);
  }
```

```
    static void addTwo(int i) {
      i += 2;
    }
  }
```

```
public class Passing {
  public static void main(String[] args) {
    int a = 0; int b = 0;
    int[] bArr = new int[1]; bArr[0] = b;

    inc1(a); inc2(bArr);

    System.out.println("a=" + a + " b=" + b + " bArr[0]=" + bArr[0]);
  }

  public static void inc1(int x) { x++; }

  public static void inc2(int[] x) { x[0]++; }
}
```

```
public class RQ810A40 {
  static void print(Object... obj) {
    System.out.println("Object...: " + obj[0]);
  }
  public static void main(String[] args) {
    // (1) INSERT METHOD CALL HERE
  }
}
```

```
public class Test {
  public static int add(int x, int y)  {
    var z = x + y;     // (1)
    return z;
  }
  public static void main(String[] args) {
    var a = 2, b = 3;  // (2)
    var z = add(a,b);  // (3)
  }
}
```

```
public class TestMe {
  /* (1) INSERT METHOD HEADER HERE */ {
    return (double)x / y;
  }
  public static void main(String[] args) {
    double x = divide(2, 3);
  }
}
```

```
if (emergency)            // emergency is a boolean variable
  operate();

if (temperature > critical)
  soundAlarm();
```

```
if (isLeapYear() && endOfCentury())
  celebrate();

if (catIsAway()) {         // Block
  getFishingRod();
  goFishing();
}
```

```
if (emergency); // Empty if block
  operate();    // Executed regardless of whether it was an emergency
```

```
if (emergency)
  operate();
else
  joinQueue();

if (temperature > critical)
  soundAlarm();
else
  businessAsUsual();

if (catIsAway()) {
  getFishingRod();
  goFishing();
} else
  playWithCat();
```

```
if (temperature >= upperLimit) {        // (1)
  if (danger)                           // (2) Simple if.
    soundAlarm();
  if (critical)                         // (3)
    evacuate();
  else                                  // Goes with if at (3).
    turnHeaterOff();
} else                                  // Goes with if at (1).
    turnHeaterOn();
```

```
// (A):
if (temperature > upperLimit) {          // (1) Block notation.
  if (danger) soundAlarm();              // (2)
} else                                   // Goes with if at (1).
  turnHeaterOn();

// (B):
if (temperature > upperLimit)            // (1) Without block notation.
  if (danger) soundAlarm();              // (2)
else turnHeaterOn();                     // Goes with if at (2).

// (C):
if (temperature > upperLimit)            // (1)
if (danger)                                    // (2)
  soundAlarm();
else                                           // Goes with if at (2).
  turnHeaterOn();
```

```
if (temperature >= upperLimit) {                             // (1)
  soundAlarm();
  turnHeaterOff();
} else if (temperature < lowerLimit) {                       // (2)
  soundAlarm();
  turnHeaterOn();
} else if (temperature == (upperLimit-lowerLimit)/2) {       // (3)
  doingFine();
} else                                                       // (4)
  noCauseToWorry();
```

```
public class Advice {

  private static final int LITTLE_ADVICE = 0;
  private static final int MORE_ADVICE = 1;
  private static final int LOTS_OF_ADVICE = 2;

  public static void main(String[] args) {
    dispenseAdvice(LOTS_OF_ADVICE);
  }

  public static void dispenseAdvice(int howMuchAdvice) {
    switch (howMuchAdvice) {                                    // (1)
      case LOTS_OF_ADVICE: System.out.println("See no evil.");   // (2)
      case MORE_ADVICE:    System.out.println("Speak no evil."); // (3)
      case LITTLE_ADVICE:  System.out.println("Hear no evil.");  // (4)
                           break;                               // (5)
      default:             System.out.println("No advice.");     // (6)
    }
  }
}
```

```
public class Seasons {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                       // (1) Outer
      case 12: case 1: case 2:                                  // (2)
        System.out.println("Snow in the winter.");
        break;
      case 3, 4: case 5:                                        // (3)
        System.out.println("Green grass in the spring.");
        break;
      case 6, 7, 8:                                             // (4)
        System.out.println("Sunshine in the summer.");
        break;
      case 9, 10, 11:                                           // (5)
        switch(monthNumber) { // Nested switch                     (6) Inner
          case 10:
            System.out.println("Halloween.");
            break;
          case 11:
            System.out.println("Thanksgiving.");
            break;
        } // End nested switch
        // Always printed for case constant 9, 10, 11
        System.out.println("Yellow leaves in the fall.");     // (7)
        break;
      default:
        System.out.println(monthNumber + " is not a valid month.");
    }
  }
}
```

Thanksgiving.
Yellow leaves in the fall.

```
...
case PASSED -> ++numbersPassed;
case FAILED -> ++numbersFailed;
...
```

```
...
case ALARM ->  { soundTheAlarm();
                 callTheFireDepartment(); }
...
```

```
...
default -> throw new IllegalArgumentException("Not a valid value");
...
```

```
public class SeasonsII {
  public static void main(String[] args) {
    int monthNumber = 11;
    switch(monthNumber) {                                             // (1) Outer
      case 12, 1,  2 -> System.out.println("Snow in the winter.");    // (2)
      case 3,  4,  5 -> System.out.println("Green grass in the spring.");   // (3)
      case 6,  7,  8 -> System.out.println("Sunshine in the summer.");      // (4)
      case 9, 10, 11 -> {                                             // (5)
        switch(monthNumber) { // Nested switch                           (6) Inner
          case 10 -> System.out.println("Halloween.");
          case 11 -> System.out.println("Thanksgiving.");
        }
        // Always printed for case constants 9, 10, 11:
        System.out.println("Yellow leaves in the fall.");             // (7)
      }
      default -> throw new IllegalArgumentException(monthNumber +
                                            " is not a valid month.");// (8)
    }
  }
}
```

Thanksgiving.
Yellow leaves in the fall.

```
public class SwitchingOnAString {
  public static final String MEDIUM = "Medium";       // (1)
  public static final String HOT = "Hot";             // (2a)
//public static       String HOT = "Hot";             // (2b) Not OK as case label
//public static final String HOT = new String("Hot"); // (2c) Not OK as case label
```

```
  public static void main(String[] args) {
    String spiceLevel = "Medium_Hot";
    switch (spiceLevel) {
      case "Mild",                                                        // (3)
            MEDIUM + "_" + HOT -> System.out.println("Enjoy your meal!"); // (4)
      case HOT                 -> System.out.println("Have fun!");        // (5)
      case "Suicide"           -> System.out.println("Good luck!");       // (6)
      default                  -> System.out.println("You being funny?");
    }
  }
}
```

```
enum SpiceGrade { MILD, MEDIUM, MEDIUM_HOT, HOT, SUICIDE; }   // (1)

public class SwitchingFun {
  public static void main(String[] args) {
    SpiceGrade spicing = SpiceGrade.HOT;
    switch (spicing) {                                // (2)
      case HOT ->  System.out.println("Have fun!");   // (3a) OK!
//    case SpiceGrade.HOT                             // (3b) Compile-time error!
//         -> System.out.println("Have fun!");
      case SUICIDE -> System.out.println("Good luck!");
```

```
            default -> System.out.println("Enjoy your meal!");
        }
    }
}
```

```
public class SeasonsIII {

  enum Season { WINTER, SPRING, SUMMER, FALL }              // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                  // (2)
      case 12: case 1: case 2:                              // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                // (4)
      case 3, 4: case 5:                                    // (5)
        System.out.println("Green grass in the spring.");
        yield Season.SPRING;                                // (6)
      case 6, 7, 8:                                         // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                                // (8)
      case 9, 10, 11:                                       // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                  // (10)
      default:                                              // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month.");
    };                                                      // (12)
    System.out.println(season);
  }
}
```

Yellow leaves in the fall.
FALL

```
    ...
    case 1 -> "ONE";
    case 2 -> yield "two";        // Compile-time error!
    ...
```

```
...
case ALARM     -> { soundTheAlarm();
                    callTheFireDepartment();
                    yield Status.EVACUATE; }  // OK

case ALL_CLEAR -> { yield Status.NORMAL; // Compile-time error: not last statement
                    standDown(); }       //                     in the block.
...
```

```
...
default -> throw new IllegalArgumentException("Not a valid value");
...
```

```
public class SeasonsIV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                     // (2)
      case 12, 1, 2 -> {                                      // (3)
        System.out.println("Snow in the winter.");
        yield Season.WINTER;                                  // (4)
      }
      case 3, 4, 5 -> {                                       // (5)
        System.out.println("Green grass in the spring.");
        yield Season.SPRING;                                  // (6)
      }
      case 6, 7, 8 -> {                                       // (7)
        System.out.println("Sunshine in the summer.");
        yield Season.SUMMER;                                  // (8)
      }
      case 9, 10, 11 -> {                                     // (9)
        System.out.println("Yellow leaves in the fall.");
        yield Season.FALL;                                    // (10)
      }
      default ->                                              // (11)
        throw new IllegalArgumentException(monthNumber + " not a valid month.");
    };                                                        // (12)
    System.out.println(season);
  }
}
```

Yellow leaves in the fall.
FALL

```
public class SeasonsV {
  enum Season { WINTER, SPRING, SUMMER, FALL }                   // (1)

  public static void main(String[] args) {
    int monthNumber = 11;
    Season season = switch(monthNumber) {                        // (2)
      case 12,  1,  2 -> Season.WINTER;                          // (3)
      case  3,  4,  5 -> Season.SPRING;                          // (4)
      case  6,  7,  8 -> Season.SUMMER;                          // (5)
      case  9, 10, 11 -> Season.FALL;                            // (6)
      default         -> throw new IllegalArgumentException(monthNumber +
                                                      " not a valid month.");
    };
    System.out.println(season);
  }
}
```

```
public class SeasonsVI {

  enum Season { WINTER, SPRING, SUMMER, FALL }                              // (1)
  record SeasonInfo(int month, Season season) {}                            // (2)

  public static void main(String[] args) {
    int monthNumber = 11;
    SeasonInfo seasonInfo = switch(monthNumber) {                           // (3)
      case 12,  1,  2 -> new SeasonInfo(monthNumber, Season.WINTER);        // (4)
      case  3,  4,  5 -> new SeasonInfo(monthNumber, Season.SPRING);        // (5)
      case  6,  7,  8 -> new SeasonInfo(monthNumber, Season.SUMMER);        // (6)
      case  9, 10, 11 -> new SeasonInfo(monthNumber, Season.FALL);          // (7)
      default         -> throw new IllegalArgumentException(monthNumber +
                                                           " not a valid month.");
    };
    System.out.println(seasonInfo);
  }
}
```

```
SeasonInfo[month=11, season=FALL]
```

```
public class IfTest {
  public static void main(String[] args) {
    if (true)
    if (false)
    System.out.println("a");
    else
    System.out.println("b");
  }
}
```

```
public class Switching {
  public static void main(String[] args) {
    final int iLoc = 3;
    switch (6) {
      case 1:
      case iLoc:
      case 2 * iLoc:
        System.out.println("I am not OK.");
      default:
        System.out.println("You are OK.");
      case 4:
        System.out.println("It's OK.");
    }
  }
}
```

```
char value = 3;
String result = "Unknown";
switch (value) {
  case 2,3,5,7:   result = "Prime";
  case 1,4,6,8,9: result = "Composite";
}
System.out.println(result);
```

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
};
System.out.println(result);
```

```
char value = 3;
String yield =
switch (value) {
  case 2,3,5,7:   yield "Prime";
  case 1,4,6,8,9: yield "Composite";
  default:        yield "Unknown";
};
System.out.println(yield);
```

```
char value = 3;
String result =
switch (value) {
  case 2,3,5,7   -> "Prime";
  case 1,4,6,8,9 -> "Composite";
  default: { yield "Unknown"; }
};
System.out.println(result);
```

```
public class RQ462 {
  public static void main(String[] args) {
    int price = 1;
    int discount = switch (price) {
      case 5, 1, 2 -> price - 1;
      case 4, 3, 6 -> price - 2;
      default      -> 0;
    };
    System.out.println(discount);
  }
}
```

```
while (noSignOfLife());     // Empty statement as loop body!
  keepLooking();            // Statement not in the loop body.
```

```
while (cat.isAway()) {          // (1)
  mice.play();
}

do {                            // (2)
  mice.play();
} while (cat.isAway());
```

```
for (initialization; loop_condition; update_expression)
    loop_body
```

```
initialization
while (loop_condition) {
   loop_body
   update_expression
}
```

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++)   // (1)
  sum += array[index];
```

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = array.length - 1; index >= 0; index--)
  sum += array[index];
```

```
for (int i = 0, j = 1, k = 2; ... ; ...) ...;       // (2)
for (int i = 0, String str = "@"; ... ; ...) ...;  // (3) Compile-time error
```

```
int i, j, k;                                        // Variable declaration
for (i = 0, j = 1, k = 2; ... ; ...) ...;           // (4) Only initialization
```

```
// (5) Not legal and ugly:
for (int i = 0, System.out.println("This won't do!"); flag; i++) { // Error!
  // loop body
}

// (6) Legal, but still ugly:
int i;                                       // Declaration factored out.
for (i = 0, System.out.println("This is legal!"); flag; i++) {     // OK.
  // loop body
}
```

```
// Legal usage but not recommended, as it can affect code comprehension.
int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };
for (int i = 0, j = sqMatrix[0].length - 1, asymDiagonal = 0;  // initialization
     i < sqMatrix.length;                                      // loop condition
     i++, j--)                                         // update expression
  asymDiagonal += sqMatrix[i][j];                      // loop body
```

```
for (;;) doProgramming();       // Infinite loop
```

```
int sum = 0;
int[] intArray = {12, 23, 5, 7, 19};
for (int index = 0; index < intArray.length; index++) { // (1) using for(;;) loop
  sum += intArray[index];
}
```

```
// Some one-dim arrays:
int[]     intArray =    {10, 20, 30};
Integer[] intObjArray = {10, 20, 30};
String[]  strArray =    {"one", "two"};
```

```
// Some two-dim arrays:
Object[][] objArrayOfArrays = {intObjArray, strArray};
Number[][] numArrayOfArrays = {{1.5, 2.5}, intObjArray, {100L, 200L}};
int[][]    intArrayOfArrays = {{20}, intArray, {40}};

// Iterate over a String array.
// Expression type is String[], and element type is String.
// String is assignable to Object (widening conversion).
for (Object obj : strArray) {}

// Iterate over an int array.
// Expression type is int[], and element type is int.
// int is assignable to Integer (boxing conversion)
for (Integer iRef : intArrayOfArrays[0]){}

// Iterate over an Integer array.
// Expression type is Integer[], and element type is Integer.
// Integer is assignable to int (unboxing conversion)
for (int i : intObjArray){}

// Iterate over a two-dim int array.
// Outer loop: expression type is int[][], and element type is int[].
// Inner loop: expression type is int[], and element type is int.
for (int[] row : intArrayOfArrays)
  for (int val : row) {}

// Iterate over a two-dim Number array.
// Outer loop: expression type is Number[][], and element type is Number[].
// Outer loop: Number[] is assignable to Object[] (widening conversion).
// Inner loop: expression type is Object[], and element type is Object.
for (Object[] row : numArrayOfArrays)
  for (Object obj : row) {}

// Outer loop: expression type is Integer[][], and element type is Integer[].
// Outer loop: Integer[] is assignable to Number[].
// Inner loop: expression type is int[], and element type is int.
// Inner loop: int is assignable to double.
for (Number[] row : new Integer[][] {intObjArray, intObjArray, intObjArray})
  for (double num : intArray) {}
```

```
// Expression type is Number[][], and element type is Number[].
// Number[] is not assignable to Number.
for (Number num : numArrayOfArrays) {}       // Compile-time error!

// Expression type is Number[], and element type is Number.
// Number is not assignable to int.
for (int row : numArrayOfArrays[0]) {}       // Compile-time error!

// Outer loop: expression type is int[][], and element type is int[].
// int[] is not assignable to Integer[].
for (Integer[] row : intArrayOfArrays)       // Compile-time error!
  for (int val : row) {}
```

```
// Expression type is Object[][], and element type is Object[].
// Object[] is not assignable to Integer[].
for (Integer[] row : objArrayOfArrays) {}    // Compile-time error!

// Outer loop: expression type is String[], and element type is String.
// Inner loop: expression type is String, which is not legal here. Not an array.
for (String str : strArray)
  for (char val : str) {}                    // Compile-time error!
```

```
L1: if (i > 0) {
  L1: System.out.println(i);    // (1) Not OK. Label L1 redeclared.
}

L1: while (i < 0) {             // (2) OK.
  L2: System.out.println(i);
}

L1: {                           // (3) OK. Labeled block.
  int j = 10;
  System.out.println(j);
}

L1: try {                       // (4) OK. Labeled try-catch-finally block.
  int j = 10, k = 0;
  L2: System.out.println(j/k);
} catch (ArithmeticException ae) {
  L3: ae.printStackTrace();
} finally {
  L4: System.out.println("Finally done.");
}
```

```
LabelA: LabelB: System.out.println("Multiple labels. Use judiciously.");
```

```
L0: int i = 0;                       // Compile-time error!
```

```
break;                  // the unlabeled form
break label;            // the labeled form
```

```
class BreakOut {

  public static void main(String[] args) {
    System.out.println("i    sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4)
        break;                              // (1) Terminate loop. Control to (2).
      // Rest of loop body skipped when i gets the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
    } // end for
    // (2) Continue here.
    int n = 2;
    switch (n) {
      case 1:
        System.out.println(n);
        break;
      case 2:
        System.out.println("Inner for(;;) loop: ");
        for (int j = 0; j <= n; j++) {
          if (j == 2)
            break;                          // (3) Terminate loop. Control to (4).
          System.out.println("j = " + j);
        }
      default:
        System.out.println("default: n = " + n); // (4) Continue here.
    }
  }
}
```

```
i    sqrt(i)
1    1.00
2    1.41
3    1.73
Inner for(;;) loop:
j = 0
j = 1
default: n = 2
```

```
out:                          // Label.
{                             // (1) Labeled block.
  // ...
  if (j == 10) break out;     // (2) Terminate block. Control to (3).
  System.out.println(j);      // Rest of the block not executed if j == 10.
  // ...
}
// (3) Continue here.
```

```
class LabeledBreakOut {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) break;         // (3) Terminate inner loop. Control to (5).
          System.out.println("Element[" + i + ", " + j + "]: " +
                                    squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) break outer; // (4) Terminate both loops. Control to (6).
        } // end inner loop
        // (5) Continue with update expression in the outer loop header.
      } // end outer loop
    // (6) Continue here.
    System.out.println("sum: " + sum);
  }
}
```

```
continue;                // the unlabeled form
continue label;          // the labeled form
```

```
class Skip {
  public static void main(String[] args) {
    System.out.println("i    sqrt(i)");
    for (int i = 1; i <= 5; ++i) {
      if (i == 4) continue;             // (1) Control to (2).
      // Rest of loop body skipped when i has the value 4.
      System.out.printf("%d    %.2f%n", i, Math.sqrt(i));
      // (2) Continue with update expression in the loop header.
    } // end for
  }
}
```

```
class LabeledSkip {
  public static void main(String[] args) {
    int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
    int sum = 0;
    outer: for (int i = 0; i < squareMatrix.length; ++i){   // (1) label
        for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
          if (j == i) continue;                             // (3) Control to (5).
          System.out.println("Element[" + i + ", " + j + "]: " +
              squareMatrix[i][j]);
          sum += squareMatrix[i][j];
          if (sum > 10) continue outer;                     // (4) Control to (6).
          // (5) Continue with update expression in the inner loop header.
        } // end inner loop
        // (6) Continue with update expression in the outer loop header.
```

```
      } // end outer loop
    System.out.println("sum: " + sum);
  }
}
```

```
public class ReturnDemo {

  public static void main (String[] args) { // (1) void method can use return.
    if (args.length == 0) return;
    output(checkValue(args.length));
  }

  static void output(int value) {  // (2) void method need not use return.
    System.out.println(value);
    return 'a';                    // Not OK. Cannot return a value.
  }

  static int checkValue(int i) {   // (3) Non-void method: Any return statement
                                   //     must return a value.
    if (i > 3)
      return i;                    // OK.
    else
      return 2.0;                  // Not OK. double not assignable to int.
  }

  static int absentMinded() {      // (4) Non-void method.
    throw new RuntimeException();  // OK: No return statement provided, but
                                   // method terminates by throwing an exception.
  }
}
```

```
class MyClass {
  public static void main(String[] args) {
    boolean b = false;
    int i = 1;
    do {
      i++;
      b = ! b;
    } while (b);
    System.out.println(i);
  }
}
```

```
public class StillMyClass {
  public static void main(String[] args) {
    int i = 0;
    int j;
    for (j = 0; j < 10; ++j) { i++; }
    System.out.println(i + " " + j);
  }
}
```

```
class AnotherClass {
  public static void main(String[] args) {
    int i = 0;
    for (; i < 10; i++) ;          // (1)
    for (i = 0;; i++) break;       // (2)
    for (i = 0; i < 10;) i++;      // (3)
    for (;;) ;                     // (4)
  }
}
```

```
outer:
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 2; j++) {
    if (i == j) {
      continue outer;
    }
    System.out.println("i=" + i + ", j=" + j);
  }
}
```

```
public class RQ400A10 {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    int sum = 0;
    for (int i : nums)
      sum += i;
    System.out.println(sum);
  }
}
```

```
public class RQ400A30 {
  public static void main(String[] args) {
    doIt();
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  for (int sum = 0, i : nums)
    sum += i;
  System.out.println(sum);
}
```

```
public static void doIt() {
  for (int sum = 0, i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

```
public static void doIt() {
  int sum = 0;
  for (int i : {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

```
public static void doIt() {
  int sum = 0;
  for (int i : new int[] {20, 30, 40})
    sum += i;
  System.out.println(sum);
}
```

```
public static void doIt() {
  int[] nums = {20, 30, 40};
  int sum = 0;
  for (int i : nums)
    sum += i;
  System.out.println(sum);
}
```

```
for (type variable : expression) statement
```

```
class TubeLight extends Light { ... }   // TubeLight is a subclass of Light.
```

```
// File: Utility.java
class Light {                           // (1)
  // Instance fields:
            int     noOfWatts;          // Wattage
  private   boolean indicator;          // On or off
  protected String  location;           // Placement

  // Static field:
  private static int counter;           // Number of Light objects created

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    ++counter;                          // Increment counter.
  }

  // Instance methods:
  public  void    switchOn()  { indicator = true; }
  public  void    switchOff() { indicator = false; }
  public  boolean isOn()      { return indicator; }
  private String  getLocation() { return location; }

  // Static methods:
  public static int getCount() { return counter; }
}
//______________________________________________________________________________
class TubeLight extends Light {         // (2) Subclass uses the extends clause.
  // Instance fields:
  private int tubeLength;               // Length in millimeters
  private int tubeDiameter;             // Diameter in millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in superclass.
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }
  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("From the subclass:");
    System.out.println("Tube length (mm): "  + getTubeLength());
    System.out.println("Tube diameter (mm): " + tubeDiameter);
    System.out.println();
    System.out.println("From the superclass:");
    System.out.println("Wattage: "       + noOfWatts);    // Inherited.
//  System.out.println("Indicator: "     + indicator);    // (3) Not inherited.
    System.out.println("Location: "      + location);     // Inherited.
//  System.out.println("Counter: "   + counter);          // (4) Not inherited.
    switchOn();                                           // Inherited
    switchOff();                                          // Inherited
    System.out.println("Indicator: "     + isOn());       // Inherited.
//  System.out.println("Location: " + getLocation());     // (5) Not inherited.
    System.out.println("Number of lights: " + getCount());// Inherited.
  }
}
//______________________________________________________________________________
public class Utility {
  public static void main(String[] args) {
    TubeLight loftLight = new TubeLight(18, true, "Loft", 590, 26);
    loftLight.printInfo();
  }
}
```

```
From the subclass:
Tube length (mm): 590
Tube diameter (mm): 26

From the superclass:
Wattage: 18
Location: Loft
Indicator: false
Number of lights: 1
```

```
Light light = new TubeLight(36, false, "Basement",
                            1200, 26);      // (1) widening reference conversion
```

```
light.switchOn();                            // (2)
```

```
    light.getTubeLength();                         // (3) Compile-time error!
```

```
// The overridden method in the superclass Light:
protected double energyCost(int noOfHours)      // (2) Instance method
    throws InvalidHoursException { ... }

// The overriding method in the subclass TubeLight:
@Override
public double energyCost(final int noOfHours)   // (7) Overriding instance
    throws ZeroHoursException { ... }           //     method at (2).
    tubeLight.energyCost(50);                    // (15) Invokes method at (7).
    light1.energyCost(50);                       // (16) Invokes method at (7).
    light2.energyCost(50);                    // (17) Invokes method at (2).
```

```
// File: Client2.java
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                  // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                       // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {          // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//______________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//______________________________________________________________________________
```

```
public class Client2 {
  public static void main(String[] args)          // (11)
      throws InvalidHoursException {

    TubeLight tubeLight = new TubeLight();        // (12)
    Light     light1    = tubeLight;              // (13) Aliases.
    Light     light2    = new Light();            // (14)

    System.out.println("Invoke overridden instance method:");
    tubeLight.energyCost(50);                     // (15) Invokes method at (7).
    light1.energyCost(50);                        // (16) Invokes method at (7).
    light2.energyCost(50);                        // (17) Invokes method at (2).

    System.out.println(
        "\nInvoke overridden instance method with covariant return:");
    System.out.println(
        light2.makeInstance().getClass());        // (18) Invokes method at (3).
    System.out.println(
        tubeLight.makeInstance().getClass());     // (19) Invokes method at (9).

    System.out.println("\nAccess hidden field:");
    System.out.println(tubeLight.lightType);      // (20) Accesses field at (6).
    System.out.println(light1.lightType);         // (21) Accesses field at (1).
    System.out.println(light2.lightType);         // (22) Accesses field at (1).

    System.out.println("\nInvoke hidden static method:");
    tubeLight.printLightType();                   // (23) Invokes method at (10).
    light1.printLightType();                      // (24) Invokes method at (5).
    light2.printLightType();                      // (25) Invokes method at (5).

    System.out.println("\nInvoke overloaded method:");
    tubeLight.energyCost();                       // (26) Invokes method at (8).
  }
}
```

```
Invoke overridden instance method:
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> Light.energyCost(int): Energy cost for Generic Light: 10.0

Invoke overridden instance method with covariant return:
>> Light.makeInstance(): class Light
>> TubeLight.makeInstance(): class TubeLight

Access hidden field:
Tube Light
Generic Light
Generic Light

Invoke hidden static method:
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
>> Static Light.printLightType(): Generic Light
```

```
Invoke overloaded method:
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
```

```
// The overridden method in the superclass Light:
public Light makeInstance() { ... }     // (3) Instance method

// The overriding method in the subclass TubeLight:
@Override
public TubeLight makeInstance() { ... } // (9) Overriding instance method at (3).
public class Overloader {

  static final void callMe(long x, Integer y) {       // (1) Overloaded by (2)
    System.out.println("long, Integer");
  }
  static void callMe(Integer x, long y) {       // (2) Overloaded by (1)
    System.out.println("Integer, long");
  }

  public static void main(String[] args) {
//  callMe(20, 17);                // (3) Ambiguous call: Box 1st or 2nd argument?
    callMe(20, Integer.valueOf(17));   // (4) Calls (1): long, Integer
    callMe(Integer.valueOf(20), 17);   // (5) Calls (2): Integer, long
  }
}
```

```
// File: Client3.java
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                        // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {           // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//______________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
  }
}
//________________________________________________________________________________
class NeonLight extends TubeLight {
  // ...
  public void demonstrate()                         // (11)
      throws InvalidHoursException {
    super.showSign();                               // (12) Invokes method at (4)
    super.energyCost(50);                           // (13) Invokes method at (7)
    super.energyCost();                             // (14) Invokes method at (8)

    ((Light) this).energyCost(50);                  // (15) Invokes method at (7)

    System.out.println(super.lightType);            // (16) Accesses field at (6)
    System.out.println(((Light) this).lightType); // (17) Accesses field at (1)

    super.printLightType();                         // (18) Invokes method at (10)
    ((Light) this).printLightType();                // (19) Invokes method at (5)
  }
}
//________________________________________________________________________________
public class Client3 {
  public static void main(String[] args)
      throws InvalidHoursException {
    NeonLight neonRef = new NeonLight();
    neonRef.demonstrate();
  }
}
```

```
>> Light.showSign(): Let there be light!
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
Tube Light
Generic Light
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
```

```
// File: DemoConstructorCall.java
class Light {
  // Fields:
  private int     noOfWatts;      // wattage
  private boolean indicator;      // on or off
  private String  location;       // placement

  // Constructors:
  Light() {                                  // (1) No-argument constructor
    noOfWatts = 0;
    indicator = false;
    location  = "X";
    System.out.println("Returning from no-argument constructor no. 1.");
  }
  Light(int watts, boolean onOffState) {                      // (2)
    noOfWatts = watts;
    indicator = onOffState;
    location  = "X";
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//______________________________________________________________________________
public class DemoConstructorCall {
  public static void main(String[] args) {                    // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);
    System.out.println("Creating Light object no. 3.");
    Light light3 = new Light(250, true, "attic");
  }
}
```

```
Creating Light object no. 1.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 2.
Creating Light object no. 3.
Returning from constructor no. 3.
```

```
// File: DemoThisCall.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;
  // Constructors:
  Light() {                                 // (1) No-argument constructor
    this(0, false);
    System.out.println("Returning from no-argument constructor no. 1.");
  }
  Light(int watt, boolean ind) {                             // (2)
    this(watt, ind, "X");
    System.out.println("Returning from constructor no. 2.");
  }
  Light(int noOfWatts, boolean indicator, String location) { // (3)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println("Returning from constructor no. 3.");
  }
}
//______________________________________________________________________
public class DemoThisCall {
  public static void main(String[] args) {                   // (4)
    System.out.println("Creating Light object no. 1.");
    Light light1 = new Light();                              // (5)
    System.out.println("Creating Light object no. 2.");
    Light light2 = new Light(250, true);                     // (6)
    System.out.println("Creating Light object no. 3.");
    Light light3 = new Light(250, true, "attic");            // (7)
  }
}
```

```
Creating Light object no. 1.
Returning from constructor no. 3.
Returning from constructor no. 2.
Returning from no-argument constructor no. 1.
Creating Light object no. 2.
Returning from constructor no. 3.
Returning from constructor no. 2.
Creating Light object no. 3.
Returning from constructor no. 3.
```

```
// File: ChainingConstructors.java
class Light {
  // Fields:
  private int     noOfWatts;
  private boolean indicator;
  private String  location;

  // Constructors:
  Light() {                                  // (1) No-argument constructor
    this(0, false);
    System.out.println(
    "Returning from no-argument constructor no. 1 in class Light");
  }
  Light(int watt, boolean ind) {                              // (2)
    this(watt, ind, "X");
    System.out.println(
    "Returning from constructor no. 2 in class Light");
  }
  Light(int noOfWatts, boolean indicator, String location) {  // (3)
    super();                                                  // (4)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    System.out.println(
        "Returning from constructor no. 3 in class Light");
  }
}
//______________________________________________________________________________
class TubeLight extends Light {
  // Instance variables:
  private int tubeLength;
  private int colorNo;

  // Constructors:
  TubeLight(int tubeLength, int colorNo) {                    // (5)
    this(tubeLength, colorNo, 100, true, "Unknown");
    System.out.println(
           "Returning from constructor no. 1 in class TubeLight");
  }
  TubeLight(int tubeLength, int colorNo, int noOfWatts,
            boolean indicator, String location) {             // (6)
    super(noOfWatts, indicator, location);                    // (7)
    this.tubeLength = tubeLength;
    this.colorNo    = colorNo;
    System.out.println(
           "Returning from constructor no. 2 in class TubeLight");
  }
}
//______________________________________________________________________________
public class ChainingConstructors {
  public static void main(String[] args) {
    System.out.println("Creating a TubeLight object.");
    TubeLight tubeLightRef = new TubeLight(20, 5);            // (8)
  }
}
```

```
Creating a TubeLight object.
Returning from constructor no. 3 in class Light
Returning from constructor no. 2 in class TubeLight
Returning from constructor no. 1 in class TubeLight
```

```
class A {
  A() {}                    // No-argument constructor.
  // ...
}
```

```
class B extends A {   // No constructors.
  // ...
}
```

```
class A {
  A() { super(); }    // (1) Call to no-argument superclass constructor inserted.
  // ...
}

class B extends A {
  B() { super(); }    // (2) Default constructor inserted.
  // ...
}
```

```
class NeonLight extends TubeLight {
  // Field
  private String sign;

  NeonLight() {                                 // (1)
    super(10, 2, 100, true, "Roof-top");    // (2) Cannot be commented out.
    sign = "All will be revealed!";
  }
  // ...
}
```

```
// Classes
class Foo {
  private int i;
  public void f() { /* ... */ }
  public void g() { /* ... */ }
}

class Bar extends Foo {
  public int j;
  public void g() { /* ... */ }
}

public class Main {
  public static void main(String[] args) {
    Foo a = new Bar();
    Bar b = new Bar()
    // (1) INSERT STATEMENT HERE
  }
}
```

```
class A {
  void doIt() {}
}

class B extends A {
  void doIt() {}
}

class C extends B {
  void doIt() {}
  void callUp() {
    // (1) INSERT EXPRESSION HERE
  }
}
```

```
class A {
  int max(int x, int y) { (x>y) ? x : y; }
}

class B extends A {
  int max(int x, int y) { return super.max(y, x) - 10; }
}

class C extends B {
  int max(int x, int y) { return super.max(x+10, y+10); }

}

public class UserClass {
  public static void main(String[] args) {
    B b = new C();
    System.out.println(b.max(13, 29));
  }
}
```

```
class Vehicle {
  static public String getModelName() { return "Volvo"; }
  public long getRegNo() { return 12345; }
}

class Car extends Vehicle {
  static public String getModelName() { return "Toyota"; }
  public long getRegNo() { return 54321; }
}

public class TakeARide {
  public static void main(String[] args) {
    Car c = new Car();
    Vehicle v = c;

    System.out.println("|" + v.getModelName() + "|" + c.getModelName() +
                       "|" + v.getRegNo()     + "|" + c.getRegNo() + "|");
  }
}
```

```
class MySuper {
  int number;
  MySuper(int i) { number = i; }
}

class MySub extends MySuper {
  int count;
  MySub(int count, int num) {
    super(num);
    this.count = count;
  }

  // (1) INSERT CONSTRUCTOR HERE
}
```

```
public class MyClass {
  public static void main(String[] args) {
    B b = new B("Test");
  }
}

class A {
  A() { this("1", "2"); }

  A(String s, String t) { this(s + t); }

  A(String s) { System.out.println(s); }
}

class B extends A {
  B(String s) { System.out.println(s); }

  B(String s, String t) { this(t + s + "3"); }

  B() { super("4"); };
}
```

```
abstract class Light {                                   // (1) Abstract class
  //...
  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)    // (2) Method header
      throws InvalidHoursException;                      // No method body
}
```

```
Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time error!
```

```
class TubeLight extends Light {
  // ...
  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
```

```
TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6)
cellarLight.switchOff();                      // Method in superclass
System.out.println(cellarLight.isOn());       // Method in superclass: false
System.out.printf("Energy cost ($): %2.2f%n",
    cellarLight.energyCost(40));              // (7) Using subclass reference
```

```
Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8)
System.out.printf("Energy cost ($): %2.2f%n",
    nightLight.energyCost(30));            // (9) Using superclass reference
                                           // Invokes method in subclass
                                           // Requires throws clause at (4)
```

```
abstract class Art { abstract void paint(); }            // (1) Abstract class

abstract class AbstractArt extends Art {}                // (2) Must be abstract

class MinimalistArt extends AbstractArt {                // (3) Concrete class
  @Override void paint() { System.out.println(":-)"); }  // (4) Concrete method
}

abstract class PostModernMinimalistArt
                                 extends MinimalistArt { // (5) Abstract class
  @Override void paint() { System.out.println(":-("); }  // (6) Concrete method
                                                         //     overrides (4)
}

class ArtsyFartsy extends PostModernMinimalistArt {}     // (7) Concrete class
```

```
// File: Factory.java
// Checked exceptions:
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

abstract class Light {                                     // (1) Abstract class
  // Fields:
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  // Abstract instance method:
  protected abstract double energyCost(int noOfHours)      // (2) Method header
      throws InvalidHoursException;                         // No method body
}
//________________________________________________________________________________
class TubeLight extends Light {
  // Instance fields:
  private int tubeLength;                                   // millimeters
  private int tubeDiameter;                                 // millimeters

  // Non-zero argument constructor
  TubeLight(int noOfWatts, boolean indicator, String location,
            int tubeLength, int tubeDiameter) {
    super(noOfWatts, indicator, location);  // Calling constructor in superclass.
    this.tubeLength = tubeLength;
    this.tubeDiameter = tubeDiameter;
  }

  // Implementation of the abstract method from the superclass.
  @Override public double energyCost(int noOfHours) {      // (3)
    return  0.15 * noOfHours;
  }
}
//________________________________________________________________________________
public class Factory {
  public static void main(String[] args) throws InvalidHoursException {  // (4)
//  Light porchLight = new Light(21, true, "Porch");    // (5) Compile-time error!
    TubeLight cellarLight = new TubeLight(18, true, "Cellar", 590, 26);  // (6)
    cellarLight.switchOff();
    System.out.println(cellarLight.isOn());        // false
    System.out.printf("Energy cost ($): %2.2f%n",
        cellarLight.energyCost(40));               // (7) Using subclass reference
    Light nightLight = new TubeLight(15, false, "Bedroom", 850, 15);     // (8)
    System.out.printf("Energy cost ($): %2.2f%n",
        nightLight.energyCost(30));             // (9) Using superclass reference
                                                // Invokes method in subclass
                                                // Requires throws clause in (4)
  }
}
```

access_modifier `abstract` *return_type method_name* (*formal_parameter_list*)
throws_clause;

```
protected abstract double energyCost(int noOfHours)      // (2) Method header
    throws InvalidHoursException;                        // No method body
```

```
@Override public double energyCost(int noOfHours) {      // (3)
  return  0.15 * noOfHours;
}
```

```
public double energyCost() {            // Overloaded
  return 1.75;
}
```

```
@Override
double energyCost(int numOfHours) {   // Not overridden! Narrows accessibility!
  return 2.0 * numOfHours;
}

public Double energyCost(int numOfHours) {  // Not overloaded! Duplicate method!
  return 3.5 * numOfHours;
}
```

```
final class TubeLight extends Light {                // (12)
  // ...
}

class NeonLight extends TubeLight {                  // Compile-time error!
  // ...
}
```

```
class Light {                                    // (1)
  // ...
  public final void setWatts(int watt) {         // (10) Final instance method
    noOfWatts = watt;
  }
}
```

```
class TubeLight extends Light {                  // (12)
  // ...
  @Override
  public void setWatts(int watt) {   // (14) Cannot override final method at (10)!
    noOfWatts = 2*watt;
  }
}
```

```
import java.util.Arrays;
import static java.lang.System.out;

class Light {                                      // (1)
  // Static final variables:
  public static final double KWH_PRICE = 3.25;   // (2) Constant static variable
  public static final String MANUFACTURER;       // (3) Blank final static field

  static {                                         // Static initializer block
    MANUFACTURER = "Ozam";                         // (4) Initializes (3)
  }

  // Instance variables:
  int noOfWatts;                                   // (5)
  final String color;                              // (6) Blank final instance field
  final String energyRating;                       // (7) Blank final instance field

  {                                                // Instance initializer block
    color = "off white";                           // (8) Initializes (6)
  }

  // Constructor:
  Light() {
    energyRating = "A++";                          // (9) Initializes (7)
  }

  public final void setWatts(int watt) {          // (10) Final instance method
    this.noOfWatts = watt;
  }
  public static void setKWH(double rate) {
//  KWH_PRICE = rate;                            // (11) Not OK. Final field.
  }
}
//______________________________________________________________________________
class TubeLight extends Light {                  // (12)
  static StringBuilder color = new StringBuilder("green");      // (13) Hiding (6)

//@Override
//public void setWatts(int watt) {   // (14) Cannot override final method at (10)!
//  noOfWatts = 2*watt;
//}
}
//______________________________________________________________________________
public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variable.
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final reference.

    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
//  carLight.setWatts(10);               // (23) Not OK. Not initialized.

    out.println("Accessing final static fields in class Light:");
    out.println("KWH_PRICE:    " + Light.KWH_PRICE);
    out.println("MANUFACTURER: " + Light.MANUFACTURER);

    out.println("Accessing final instance fields in an object of class Light:");
    out.println("noOfWatts:    " + workLight.noOfWatts);
    out.println("color:        " + workLight.color);
    out.println("energyRating: " + workLight.energyRating);

    out.println("Fun with final parameter args:");
    out.println(Arrays.toString(args));  // Print array.
    out.println("args length: " + args.length);
    args[0] = "1";                       // (24) OK. Modifying array state.
    out.println(Arrays.toString(args));  // Print array.
//  args = null;                         // (25) Not OK. Final parameter.
//  args.length = 10;                    // (26) Not OK. Final instance field.
  }
}
```

```
>java Warehouse One Two Three
Accessing final static fields in class Light:
KWH_PRICE:    3.25
MANUFACTURER: Ozam
Accessing final instance fields in an object of class Light:
noOfWatts:    100
color:        off white
```

```
energyRating: A++
Fun with final parameter args:
[One, Two, Three]
args length: 3
[1, Two, Three]
```

```
class Light {                                     // (1)
  public static final double KWH_PRICE = 3.25;   // (2) Constant static variable
  public static final String MANUFACTURER;       // (3) Blank final static field

  int noOfWatts;                                  // (5) Non-final instance field
  final String color;                             // (6) Blank final instance field
  final String energyRating;                      // (7) Blank final instance field
  // ...
}
```

```
public class Warehouse {
  public static void main(final String[] args) { // (15) Final parameter

    final Light workLight = new Light(); // (16) Non-blank final local variable.

    final Light alarmLight;              // (20) Blank final local variable.

    Light carLight;                      // (22) Non-final local variable.
    // ...
  }
}
```

```
    final Light workLight = new Light(); // (16) Final local variable.
    workLight.setWatts(100);             // (17) OK. Changing object state.
//  workLight.color = "pink";            // (18) Not OK. Final instance field.
//  workLight = new Light();             // (19) Not OK. Changing final reference.

    final Light alarmLight;              // (20) Blank final local variable.
//  alarmLight.setWatts(200);            // (21) Not OK. Not initialized.

    Light carLight;                      // (22) Non-final local variable.
//  carLight.setWatts(10);               // (23) Not OK. Not initialized.
```

```
public static void main(final String[] args) {  // (15) Final parameter
  // ...
  out.println("Fun with final parameter args:");
  out.println(Arrays.toString(args));         // (24) Print array.
  out.println("args length: " + args.length); // (25) Access final field
  args[0] = "1";                              // (26) OK. Modifying array state.
  out.println(Arrays.toString(args));         // Print array.
//args = null;                                // (27) Not OK. Final parameter.
//args.length = 10;                          // (28) Not OK. Final instance field.
}
```

```
final int k;              // Declaration: blank final local variable
k = 10;                   // (1) Assignment
System.out.println(k);    // (2) Access: k is definitely assigned.

boolean status = true;    // Non-constant variable.
final int j;              // Declaration: blank final local variable
if (status) {             // (3) Value of conditional expression not evaluated
  j = 5;                  // (4) Conditional assignment
}
System.out.println(j);    // (5) Access: j is not definitely assigned!
```

```
int n = 5;
final int k;            // Declaration: blank final local variable
if (n >= 4) {           // (1) Value of conditional expression not evaluated
  k = 6;                // (2) Conditional assignment: k is definitely unassigned
}
k = 12;                 // (3) Assignment: k is not definitely unassigned!
System.out.println(k);  // (4) Access: k is definitely assigned
```

```
int n = 5;
final int k;             // Declaration: blank final local variable
if (n >= 4) {            // (1) Value of conditional expression not evaluated
  k = 6;                 // (2) Conditional assignment: k is definitely unassigned
} else {
  k = 12;                // (3) Conditional assignment: k is definitely unassigned
}
System.out.println(k);  // (4) Access: k is definitely assigned
```

```
{
  final int i = 10;
  i++;               // Not OK. Side effect changes the value of i.
}

{
  for (int i = 0; i < 10; i++) {
    final int j = i; // OK. Final variable goes out of scope after each iteration.
  }
}

{
  final int j;
  for (int i = 0; i < 10; i++) {
    j = i;  // Not OK. Loop either not executed and j not initialized
            // or each iteration will assign a new value to j.
  }
  System.out.println(j); // Not OK. Not guaranteed that j is initialized.
}
```

```
class Ghost          { abstract void haunt(); }
```

```
abstract class Ghost { void haunt(); }
abstract class Ghost { void haunt() {}; }
abstract Ghost       { abstract void haunt(); }
abstract class Ghost { abstract haunt(); }
```

```
public class Animal {
  // (1)
}
```

```
public abstract class Cat extends Animal {
  // (2)
}
```

```
public abstract void eat();
```

```
abstract class LivingOrganism {
  public abstract void feed();
}

class Bacteria extends LivingOrganism {
  @Override
  public void feed()  {  }
  public void feed(int quantity)  {  }        // (1)
}

public class Lab {
  public static void main(String[] args) {
    LivingOrganism org = new Bacteria();      // (2)
    org.feed();
    ((Bacteria)org).feed(10);                 // (3)
  }
}
```

```
abstract class MyClass {
  abstract void f();
  final    void g() {}
//final    void h() {}                                    // (1)

  protected static int i;
  private          int j;
}

final class MyOtherClass extends MyClass {
//MyOtherClass(int n) { m = n; }                          // (2)

  public static void main(String[] args) {
    MyClass mc = new MyOtherClass();
  }

  void f() {}
  void h() {}
//void k() { i++; }                                       // (3)
//void l() { j++; }                                       // (4)

  int m;
}
```

```
public abstract class Animal {
  public static final MAX_SIZE = 10;
  public abstract void measure(int size);
}

public class Cat extends Animal {
  private int size;
  @Override
  public final void measure(int size) {
    this.size = (size < Animal.MAX_SIZE) ? size : Animal.MAX_SIZE;
  }
}
```

```
access_modifier interface interface_name
                              optional_type_parameter_list
                              optional_extends_interface_clause // Interface header
{ // Interface body
    abstract_method_declarations
    default_method_declarations
    static_method_declarations
    private_instance_method_declarations
    private_static_method_declarations
    constant_declarations
    member_type_declarations
}
```

return_type method_name (*formal_parameter_list*) *throws_clause*;

```
interface Playable {
  void play();              // Abstract method: no implementation
}
```

```
// File: StackUser.java
interface IStack {                                                        // (1)
  void   push(Object item);
  Object pop();
}
//_________________________________________________________________________
class Stack implements IStack {                                           // (2)
  protected Object[] elements;
  protected int      tos;  // top of stack

  public Stack(int capacity) {
    elements = new Object[capacity];
    tos        = -1;
  }

  @Override
  public void push(Object item) { elements[++tos] = item; }               // (3)

  @Override
  public Object pop() {                                                   // (4)
    Object objRef = elements[tos];
    elements[tos] = null;
    tos--;
    return objRef;
  }
  public Object peek() { return elements[tos]; }
}
//______________________________________________________________________________
interface ISafeStack extends IStack {                                     // (5)
  boolean isEmpty();
  boolean isFull();
}
//______________________________________________________________________________
class SafeStack extends Stack implements ISafeStack {                     // (6)

  public SafeStack(int capacity) { super(capacity); }
  @Override public boolean isEmpty() { return tos < 0; }                  // (7)
  @Override public boolean isFull()  { return tos >= elements.length-1; } // (8)
}
//______________________________________________________________________________
public class StackUser {

  public static void main(String[] args) {                                // (9)
    SafeStack  safeStackRef  = new SafeStack(10);
    Stack      stackRef      = safeStackRef;
    ISafeStack isafeStackRef = safeStackRef;
    IStack     istackRef     = safeStackRef;
    Object     objRef        = safeStackRef;

    safeStackRef.push("Dollars");                                         // (10)
    stackRef.push("Kroner");
    System.out.println(isafeStackRef.pop());
    System.out.println(istackRef.pop());
    System.out.println(objRef.getClass());
  }
}
```

```
interface IA { int doIt(); }                             // (1)

interface IB { int doIt(); }                             // (2)

abstract class Combined implements IA, IB { }            // (3)

public class Worker implements IA, IB {                  // (4)
  @Override
  public int doIt() { return 0; }                        // (5)
}
```

```
interface IA { int doIt(); }                            // (1)

interface IB { double doIt(); }                         // (2)

abstract class Combined implements IA, IB { }           // (3) Compile-time error!

public class ChallengedWorker implements IA, IB {       // (4)
  @Override
  public int doIt() { return 0; }                       // (5) Compile-time error!
  @Override
  public double doIt() {                                // (6) Compile-time error!
    System.out.println("Sorry!");
    return = 0.0;
  }
}
```

```
interface IStack {                                              // (1)
  void   push(Object item);
  Object pop();
  @Override boolean equals(Object other);       // public method in Object class.
  @Override String toString();                  // public method in Object class.
//@Override Class getClass(); // Compile-time error! final method in Object class.
}
```

`default` *return_type method_name* (*formal_parameter_list*) *throws_clause* {
    implementaion_of_method_body
}

```
// File: JavaParty.java
interface ISlogan {
  default void printSlogan() {                       // (1)
    System.out.println("Happiness is getting certified!");
  }
}
//_______________________________________________________________________________
class JavaGuru implements ISlogan {
  @Override
  public void printSlogan() {                        // (2) overrides (1)
    System.out.println("Happiness is catching all the exceptions!");
  }
}
//_______________________________________________________________________________
class JavaGeek implements ISlogan { }                // (3) inherits (1)
//_______________________________________________________________________________
public class JavaParty {
  public static void main(String[] args) {
    JavaGuru guru = new JavaGuru();
    guru.printSlogan();                              // (4)
    JavaGeek geek = new JavaGeek();
    geek.printSlogan();                              // (5)
  }
}
```

```
Happiness is catching all the exceptions!
Happiness is getting certified!
```

```
interface ISlogan {
  default void printSlogan() {          // (1) Default method.
```

```
    System.out.println("Happiness is getting certified!");
  }
}

interface INewSlogan extends ISlogan {
  @Override
  abstract void printSlogan();         // (2) overrides (1) with abstract method.
}

abstract class JavaMaster implements ISlogan {
  @Override
  public abstract void printSlogan();  // (3) overrides (1) with abstract method.
}
```

```
// File: MultipleInheritance.java
interface ICheapSlogan {
  default void printSlogan() {          // (1)
    System.out.println("Override, don't overload.");
  }
}
//_______________________________________________________________________________
interface IFunnySlogan {
  default void printSlogan() {          // (2)
    System.out.println("Catch exceptions, not bugs.");
  }
}
//_______________________________________________________________________________
interface IAvailableSlogan              // (3) Compile-time error.
          extends ICheapSlogan, IFunnySlogan { }
//_______________________________________________________________________________
abstract class Wholesaler               // (4) Compile-time error.
               implements ICheapSlogan, IFunnySlogan { }
//_______________________________________________________________________________
abstract class RetailSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Abstract method.
  public abstract void printSlogan();   // (5) overrides (1) and (2).
}
//_______________________________________________________________________________
class NetSeller implements ICheapSlogan, IFunnySlogan {
  @Override                             // Concrete method.
  public void printSlogan() {           // (6) overrides (1) and (2).
    System.out.println("Think outside of the class.");
  }

  public void invokeDirect() {          // (7)
    ICheapSlogan.super.printSlogan();   // (8) calls ICheapSlogan.printSlogan()
    IFunnySlogan.super.printSlogan();   // (9) calls IFunnySlogan.printSlogan()
  }
}
//_______________________________________________________________________________
public class MultipleInheritance {      // (10)
  public static void main(String[] args) {
    NetSeller seller = new NetSeller();
    seller.printSlogan();               // (11)
    seller.invokeDirect();
  }
}
```

Think outside of the class.
Override, don't overload.
Catch exceptions, not bugs.

```
// File: MultipleInheritance2.java
class Slogan {
  public void printSlogan() {                         // (1) Concrete method
    System.out.println("Superclass wins!");
  }
}
//_______________________________________________________________________________
interface ISlogan {
  default void printSlogan() {                        // (2) Default method
    System.out.println("Superinterface wins!");
  }
}
//_______________________________________________________________________________
class MySlogan extends Slogan implements ISlogan { } // (3)
//_______________________________________________________________________________
public class MultipleInheritance2 {                  // (4)
  public static void main(String[] args) {
    MySlogan slogan = new MySlogan();
    slogan.printSlogan();                            // (5)
  }
}
```

```
static return_type method_name (formal_parameter_list) throws_clause {
    implementaion_of_method_body
}
```

```
// File: CarRace.java
import static java.lang.System.out;
```

```
interface IMaxEngineSize {
  static int getNumOfCylinders() { return 6; }        // (1) Static method
  static double getEngineSize() { return 1.6; }       // (2) Static method
}
//_______________________________________________________________________________
interface INewEngineSize extends IMaxEngineSize {
  static double getEngineSize() { return 2.4; }       // (3) Static method
}
//_______________________________________________________________________________
public class CarRace implements INewEngineSize {
  public static void main(String[] args) {
//  out.println("No. of cylinders: " +
//              getNumOfCylinders());                 // (4) Compile-time error.
    out.println("No. of cylinders: " +
        IMaxEngineSize.getNumOfCylinders());          // (5)
//  out.println("Engine size: " + getEngineSize());   // (6) Compile-time error.
    out.println("Max engine size: " + IMaxEngineSize.getEngineSize()); // (7)
    out.println("New engine size: " + INewEngineSize.getEngineSize()); // (8)
  }
}
```

```
/** Interface with private methods. */
public interface IShopper {

  // Abstract method:
  double getItemPrice();                               // (1)

  // Default method:
  default double getSalePrice(double price) {          // (2)
    var salePrice = (80.0/100.0)*price;
    System.out.println("Default method: " + "Sale price is " + salePrice);
    wrapUp();                    // (3) Calls the private instance method at (4)
    return salePrice;
  }

  // Private instance method:
  private void wrapUp() {                               // 4)
    System.out.println("Private method: " + "Wrapping up!");
  }

  // Static method:
  static void startSale() {                             // (5)
    System.out.println("Static method: " + "Amazing savings!");
    showSaleItems();             // (6) Calls the private static method at (7)
  }

  // Private static method:
  private static void showSaleItems() {                 // (7)
    System.out.println("Private static method: " + "Sorry. No items on sale!");
  }
}
```

```
/** Class Shopper implements IShopper interface */
public class Shopper implements IShopper {

  @Override
  public double getItemPrice() {
    return 100.00;
  }

  public static void main(String[] args) {
    Shopper customer = new Shopper();
    var price = customer.getItemPrice();
    System.out.println("Item price: " + price);
    var salePrice = customer.getSalePrice(price); // Calls default method at (2)
    System.out.println();
    IShopper.startSale();                        // Calls static method at (5)
  }
}
```

```
Item price: 100.0
Default method: Sale price is 80.0
Private method: Wrapping up!

Static method: Amazing savings!
Private static method: Sorry. No items on sale!
```

```
// File: Client.java
interface Constants {
  double PI_APPROXIMATION = 3.14;
  String AREA_UNITS       = "sq.cm.";
  String LENGTH_UNITS     = "cm.";
}
//______________________________________________________________________________
public class Client implements Constants {
  public static void main(String[] args) {
    double radius = 1.5;

    // (1) Using simple name:
    System.out.printf("Area of circle is %.2f %s%n",
               PI_APPROXIMATION * radius*radius, AREA_UNITS);

    // (2) Using qualified name:
    System.out.printf("Circumference of circle is %.2f %s%n",
             2.0 * Constants.PI_APPROXIMATION * radius, Constants.LENGTH_UNITS);
  }
}
```

```
Area of circle is 7.06 sq.cm.
Circumference of circle is 9.42 cm.
```

```
// File: Factory.java
import static java.lang.System.out;

interface IBaseStates {
  String IDLE = "idle";                                // (1)
  String BUSY = "busy";                                // (2)
  String BLOCKED = "blocked";                          // (3)
}
//_______________________________________________________________________________
interface IExtStatesA extends IBaseStates {
  String DISMANTLED = "dismantled";                    // (4)
}
//_______________________________________________________________________________
interface IExtStatesB extends IBaseStates {
  String DISMANTLED = "kaput";                         // (5)
}
//_______________________________________________________________________________
interface IAllStates extends IExtStatesB, IExtStatesA {
  String BLOCKED = "out of order";                     // (6) hides (3)
//String OBSOLETE = BLOCKED + ", " +
//                  DISMANTLED + " and scrapped.";     // (7) Ambiguous
  String OBSOLETE = BLOCKED + ", " +
         IExtStatesB.DISMANTLED + " and scrapped";     // (7a)
}
//_______________________________________________________________________________
public class Factory implements IAllStates {
  public static void main(String[] args) {
//  out.println("Machine A is " + DISMANTLED);             // (8) Ambiguous.
    out.println("Machine A is " + IExtStatesB.DISMANTLED); // (8a)
    out.println("Machine B is " + OBSOLETE);           // (9) IAllStates.OBSOLETE
    out.println("Machine C is " + BLOCKED);            // (10) IAllStates.BLOCKED
    out.println("Machine D is " + IBaseStates.BLOCKED);// (11)
    out.println("Machine E is " + BUSY);               // (12) Simple name
  }
}
```

```
Machine A is kaput
Machine B is out of order, kaput and scrapped
Machine C is out of order
Machine D is blocked
Machine E is busy
```

```
interface INewSlogan {
  String SLOGAN = "Trouble shared is trouble halved!";
  static void printSlogan() { System.out.println(SLOGAN); }
}
//______________________________________________________________________________
public class Firm implements INewSlogan {
  public static void main(String[] args) {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (1) INSERT STATEMENT EXPRESSION HERE
  }

  void testSlogan() {
    Firm co = new Firm();
    INewSlogan sl = co;
    // (2) INSERT STATEMENT EXPRESSION HERE
  }
}
```

```
interface IJogger {
  default boolean justDoIt(String msg) { return false; }  // (1)
  static  boolean justDoIt(int i)      { return true; }   // (2)
}
```

```
class Athlete implements IJogger {
  public boolean justDoIt(String msg)  { return true; }   // (3)
  public boolean justDoIt(int i)       { return false; }  // (4)
}

public class RaceA {
  public static void main(String[] args) {
    Athlete athlete = new Athlete();
    IJogger jogger = athlete;
    System.out.print(jogger.justDoIt("Run"));             // (5)
    System.out.println("|" + athlete.justDoIt(10));       // (6)
  }
}
```

```
abstract class MyClass implements Interface1, Interface2 {
  public void f() { }
  public void g() { }
}

interface Interface1 {
  int VAL_A = 1;
  int VAL_B = 2;

  void f();
  void g();
}

interface Interface2 {
  int VAL_B = 3;
  int VAL_C = 4;

  void g();
  void h();
}
```

```
ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
```

```
Stack[] arrayOfStack = new SafeStack[2];      // (1)
```

```
arrayOfStack[0] = new SafeStack(10);          // (2)
```

```
    arrayOfStack[1] = new Stack(20);               // (3) ArrayStoreException
```

```
Object obj = "Up the tree";    // Widening reference conversion: Object <-- String
String str1 = obj;       // Not OK. Narrowing reference conversion requires a cast.
String str2 = Integer.valueOf(10); // Illegal. No relation between
                                   //          String and Integer.
```

```
Integer iRef = 10;  // Only boxing
Number num = 10L;   // Boxing, followed by widening: Number <--- Long <--- long
Object obj = 100;   // Boxing, followed by widening: Object <--- Integer <--- int
```

```
// See Example 5.10, p. 241, for type declarations.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

```
public class ReferenceConversion {

  public static void main(String[] args) {
    // Reference declarations:
    Object     objRef;
    Stack      stackRef;
    SafeStack  safeStackRef;
    IStack     iStackRef;
    ISafeStack iSafeStackRef;

    // SourceType is a class type:
    safeStackRef  = new SafeStack(10);
    objRef        = safeStackRef;    // (1) Always possible
    stackRef      = safeStackRef;    // (2) Subclass to superclass assignment
    iStackRef     = stackRef;        // (3) Stack implements IStack
    iSafeStackRef = safeStackRef;    // (4) SafeStack implements ISafeStack

    // SourceType is an interface type:
    objRef    = iStackRef;           // (5) Always possible
    iStackRef = iSafeStackRef;       // (6) Sub- to superinterface assignment

    // SourceType is an array type:
    Object[]     objArray          = new Object[3];
    Stack[]      arrayOfStack      = new Stack[3];
    SafeStack[]  arrayOfSafeStack  = new SafeStack[5];
    ISafeStack[] arrayOfISafeStack = new ISafeStack[5];
    int[]        intArray          = new int[10];
    // Reference value assignments:
    objRef     = objArray;            // (7) Always possible
    objRef     = arrayOfStack;        // (8) Always possible
    objArray   = arrayOfStack;        // (9) Always possible
    objArray   = arrayOfISafeStack;   // (10) Always possible
    objRef     = intArray;            // (11) Always possible
    //  objArray   = intArray;        // (12) Compile-time error:
                                      //      int[] not subtype of Object[]
    arrayOfStack = arrayOfSafeStack; // (13) Subclass array to superclass array
    arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implements
                                          //      ISafeStack

    // Method invocation conversions:
    System.out.println("First call:");
    sendParams(stackRef, safeStackRef, iStackRef,
               arrayOfSafeStack, arrayOfISafeStack);                // (15)
    //  Call Signature: sendParams(Stack, SafeStack, IStack,
    //                              SafeStack[], ISafeStack[]);

    System.out.println("Second call:");
    sendParams(arrayOfISafeStack, stackRef, iSafeStackRef,
               arrayOfStack, arrayOfSafeStack);                     // (16)
    //  Call Signature: sendParams(ISafeStack[], Stack, ISafeStack,
    //                              Stack[], SafeStack[]);
  }

  public static void sendParams(Object objRefParam, Stack stackRefParam,
      IStack iStackRefParam, Stack[] arrayOfStackParam,
      IStack[] arrayOfIStackParam) {                                 // (17)
    //  Signature: sendParams(Object, Stack, IStack, Stack[], IStack[])
    //  Print class name of object denoted by the reference at runtime.
    System.out.println(objRefParam.getClass());
    System.out.println(stackRefParam.getClass());
    System.out.println(iStackRefParam.getClass());
    System.out.println(arrayOfStackParam.getClass());
    System.out.println(arrayOfIStackParam.getClass());
  }
}
```

```
First call:
class SafeStack
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
Second call:
class [LSafeStack;
class SafeStack
class SafeStack
class [LSafeStack;
class [LSafeStack;
```

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

```
objRef       = safeStackRef;    // (1) Always possible
stackRef     = safeStackRef;    // (2) Subclass to superclass assignment
iStackRef    = stackRef;        // (3) Stack implements IStack
iSafeStackRef = safeStackRef;   // (4) SafeStack implements ISafeStack
```

```
objRef    = iStackRef;     // (5) Always possible
iStackRef = iSafeStackRef; // (6) Subinterface to superinterface assignment
```

```
objRef     = objArray;           // (7) Always possible
objRef     = arrayOfStack;       // (8) Always possible
objArray   = arrayOfStack;       // (9) Always possible
objArray   = arrayOfISafeStack;  // (10) Always possible
objRef     = intArray;           // (11) Always possible
// objArray   = intArray;        // (12) Compile-time error:
                                 //      int[] not subtype of Object[]
arrayOfStack = arrayOfSafeStack; // (13) Subclass array to superclass array
arrayOfISafeStack = arrayOfSafeStack; // (14) SafeStack implements
                                      //      ISafeStack
```

`sendParams(Object, Stack, IStack, Stack[], IStack[])`

```
sendParams(Stack, SafeStack, IStack, SafeStack[], ISafeStack[]);   // Compile time
sendParams(SafeStack, SafeStack, SafeStack, SafeStack[], SafeStack[]);  // Runtime
sendParams(ISafeStack[], Stack, ISafeStack, Stack[], SafeStack[]); // Compile time
sendParams(SafeStack[], SafeStack, SafeStack, SafeStack[], SafeStack[]);// Runtime
```

```
private static void flipFlop(String str, int i, Integer iRef) { // (1)
    out.println(str + " ==> (String, int, Integer)");
}
private static void flipFlop(String str, int i, int j) {        // (2)
    out.println(str + " ==> (String, int, int)");
}
```

```
flipFlop(String, int, Integer)                       // See (1)
flipFlop(String, int, int)                           // See (2)
```

```
    flipFlop("(String, Integer, int)",
             Integer.valueOf(4), 2020);                        // (3) Ambiguous call
```

```
flipFlop(String, Integer, int)                              // See (3)
```

```
class Light { /* ... */ }

class TubeLight extends Light { /* ... */ }

public class Overload {
  boolean testIfOn(Light aLight)         { return true; }    // (1)
  boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)
  public static void main(String[] args) {

    TubeLight tubeLight = new TubeLight();
    Light     light     = new Light();
    Light     light2    = new TubeLight();

    Overload client = new Overload();
    System.out.println(client.testIfOn(tubeLight)); // (3) ==> method at (2)
    System.out.println(client.testIfOn(light));     // (4) ==> method at (1)
    System.out.println(client.testIfOn(light2));    // (5) ==> method at (2)
  }
}
```

`(String) => (String)`	(8) calls (1)
`(String, int) => (String, int)`	(9) calls (2)
`(String, Integer) => (String, int)`	(10) calls (2)
`(String, int, byte) => (String, int, int)`	(11) calls (3)
`(String, int, int) => (String, int, int)`	(12) calls (3)

`(String, int, long) => (String, Number[])`	(13) calls (5)
`(String, int, int, int) => (String, Integer[])`	(14) calls (4)
`(String, int, double) => (String, Number[])`	(15) calls (5)
`(String, int, String) => (String, Object[])`	(16) calls (6)
`(String, boolean) => (String, Object[])`	(17) calls (6)

```
import static java.lang.System.out;

class OverloadResolution {

  public void action(String str) {                  // (1)
    String signature = "(String)";
    out.println(str + " => " + signature);
  }

  public void action(String str, int m) {           // (2)
    String signature = "(String, int)";
    out.println(str + " => " + signature);
  }

  public void action(String str, int m, int n) {    // (3)
    String signature = "(String, int, int)";
    out.println(str + " => " + signature);
  }

  public void action(String str, Integer... data) { // (4)
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Number... data) {  // (5)
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }

  public void action(String str, Object... data) {  // (6)
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }

  public static void main(String[] args) {
    OverloadResolution ref = new OverloadResolution();
    ref.action("(String)");                                     // (8)  calls (1)
    ref.action("(String, int)",       10);                      // (9)  calls (2)
    ref.action("(String, Integer)",   Integer.valueOf(10)); // (10) calls (2)
    ref.action("(String, int, byte)",     10, (byte)20);    // (11) calls (3)
    ref.action("(String, int, int)",      10,  20);         // (12) calls (3)
    ref.action("(String, int, long)",     10,  20L);        // (13) calls (5)
    ref.action("(String, int, int, int)", 10,  20,  30);    // (14) calls (4)
    ref.action("(String, int, double)",   10,  20.0);       // (15) calls (5)
    ref.action("(String, int, String)",   10,  "what?");    // (16) calls (6)
    ref.action("(String, boolean)",       false);           // (17) calls (6)
  }
}
```

(*destination_type*) *reference_expression*

```
Object  obj = new String("Cast me!");
String  str = (String) obj;          // (1) Cast from Object to String.
Integer iRef1 = (Integer) obj;       // (2) Cast from Object to Integer, but
                                     //     ClassCastException at runtime.
Integer iRef2 = (Integer) str;       // (3) Compile-time error!
                                     //     Cast between unrelated types.
// (1) Boxing and casting: Number <-- Integer <-- int:
Number num = (Number) 100;
// (2) Casting, boxing, casting: Object <-- Integer <-- int <-- double:
Object obj = (Object) (int) 10.5;
// (3) Casting, unboxing, casting: double <--- int <-- Integer <-- Object:
double d = (double) (Integer) obj;
```

reference_expression `instanceof` *destination_type* *(destination_type) reference_expression*

```
// See Figure 5.1, p. 194, for inheritance hierarchy.
class Light { /* ... */ }
class LightBulb extends Light { /* ... */ }
class SpotLightBulb extends LightBulb { /* ... */ }
class TubeLight extends Light { /* ... */ }
class NeonLight extends TubeLight { /* ... */ }

public class WhoAmI {
  public static void main(String[] args) {
    boolean result1, result2, result3, result4;
    Light light1 = new LightBulb();                    // (1)
    //  String str = (String) light1;                  // (2) Compile-time error!
    //  result1 = light1 instanceof String;            // (3) Compile-time error!

    result2 = light1 instanceof TubeLight;             // (4) false: peer class.
    //  TubeLight tubeLight1 = (TubeLight) light1;     // (5) ClassCastException!

    result3 = light1 instanceof SpotLightBulb;         // (6) false: superclass.
    //  SpotLightBulb spotRef = (SpotLightBulb) light1;// (7) ClassCastException!

    light1 = new NeonLight();                          // (8)
    if (light1 instanceof TubeLight) {                 // (9) true.
      TubeLight tubeLight2 = (TubeLight) light1;       // (10) OK.
      // Can now use tubeLight2 to access an object of the class NeonLight,
      // but only those members that the object inherits or overrides
      // from the superclass TubeLight.
    }

    SpotLightBulb light2 = new SpotLightBulb();        // (11)
    result4 = light2 instanceof Light;                 // (12) true.
    Light light = (Light) light2;                      // (13) OK. Redundant cast.
  }
}
```

```
// See Figure 5.4, p. 245, for inheritance hierarchy.
interface IStack
interface ISafeStack extends IStack
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

```
public class Identification {
  public static void main(String[] args) {
    Object obj = new Object();
    Stack stack = new Stack(10);
    SafeStack safeStack = new SafeStack(5);
    IStack iStack;

    String strFormat = "(%d)  %-25s instance of %-25s: %s%n";
    System.out.printf(strFormat, 1,
        null, Object.class, null instanceof Object);      // Always false.
    System.out.printf(strFormat, 2,
        null, IStack.class, null instanceof IStack);      // Always false.

    System.out.printf(strFormat, 3, stack.getClass(), Object.class,
        stack instanceof Object);     // true: instance of subclass of Object.
    System.out.printf(strFormat, 4, obj.getClass(), Stack.class,
        obj instanceof Stack);        // false: Object not subtype of Stack.
    System.out.printf(strFormat, 5, stack.getClass(), Stack.class,
        stack instanceof Stack);      // true: instance of Stack.
    System.out.printf(strFormat, 6, obj.getClass(), IStack.class,
        obj instanceof IStack);       // false: Object does not implement IStack.
    System.out.printf(strFormat, 7, safeStack.getClass(), IStack.class,
        safeStack instanceof IStack); // true: SafeStack implements IStack.

    obj = stack;           // No cast required: assigning subclass to superclass.
    System.out.printf(strFormat, 8, obj.getClass(), Stack.class,
        obj instanceof Stack);        // true: instance of Stack.
    System.out.printf(strFormat, 9, obj.getClass(), IStack.class,
        obj instanceof IStack);       // true: Stack implements IStack.
    System.out.printf(strFormat, 10, obj.getClass(), String.class,
        obj instanceof String);       // false: no relationship.
    iStack = (IStack) obj; // Cast required: assigning superclass to subclass.
    System.out.printf(strFormat, 11, iStack.getClass(), Object.class,
        iStack instanceof Object);       // true: instance of subclass of Object.
    System.out.printf(strFormat, 12, iStack.getClass(), Stack.class,
        iStack instanceof Stack);        // true: instance of Stack.

    String[] strArray = new String[10];
//  System.out.printf(strFormat, 13, strArray.getClass(), String.class,
//      strArray instanceof String);     // Compile-time error: no relationship.
    System.out.printf(strFormat, 14, strArray.getClass(), Object.class,
        strArray instanceof Object);     // true: array subclass of Object.
    System.out.printf(strFormat, 15, strArray.getClass(), Object[].class,
        strArray instanceof Object[]);   // true: array subclass of Object[].
    System.out.printf(strFormat, 16, strArray[0], Object.class,
        strArray[0] instanceof Object);  // false: strArray[0] is null.
    System.out.printf(strFormat, 17, strArray.getClass(), String[].class,
        strArray instanceof String[]);   // true: array of String.

    strArray[0] = "Amoeba strip";
    System.out.printf(strFormat, 18, strArray[0].getClass(), String.class,
        strArray[0] instanceof String);  // true: strArray[0] instance of String.
  }
}
```

```
(1)  null                      instance of class java.lang.Object   : false
(2)  null                      instance of interface IStack         : false
(3)  class Stack               instance of class java.lang.Object   : true
(4)  class java.lang.Object    instance of class Stack              : false
(5)  class Stack               instance of class Stack              : true
(6)  class java.lang.Object    instance of interface IStack         : false
(7)  class SafeStack           instance of interface IStack         : true
(8)  class Stack               instance of class Stack              : true
(9)  class Stack               instance of interface IStack         : true
(10)  class Stack               instance of class java.lang.String   : false
(11)  class Stack               instance of class java.lang.Object   : true
(12)  class Stack               instance of class Stack              : true
(14)  class [Ljava.lang.String; instance of class java.lang.Object   : true
(15)  class [Ljava.lang.String; instance of class [Ljava.lang.Object;: true
(16)  null                      instance of class java.lang.Object   : false
(17)  class [Ljava.lang.String; instance of class [Ljava.lang.String;: true
(18)  class java.lang.String    instance of class java.lang.String   : true
```

```
interface IStack                      (defines methods push() and pop())
interface ISafeStack extends IStack (with boolean methods isEmpty() and isFull())
class Stack implements IStack
class SafeStack extends Stack implements ISafeStack
```

```
IStack stack = new SafeStack(20); // Supertype reference denotes subtype object.

// Using the instanceof type comparison operator. (1)
if (stack instanceof SafeStack) {                // Correct subtype?
  SafeStack safestack = (SafeStack) stack;       // Cast to subtype.
  System.out.println(safestack.isFull());        // Call subtype-specific method.
}

// Using the instanceof pattern match operator.   (2)
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());
}
```

```
ISafeStack safestack = new SafeStack(20);
if (safestack instanceof ISafeStack stack) {     // (1) Compile-time error!
    stack.push("Hi");
}
if (safestack instanceof IStack stack) {         // (2) Compile-time error!
    stack.push("Howdy");
}
```

```
IStack stack = new SafeStack(20);
SafeStack safestack = new SafeStack(5);       // Local variable safestack

if (stack instanceof SafeStack safestack) {   // Error: safestack redeclared.
  System.out.println(safestack.isFull());
}
```

```
// Field declaration:
private static IStack myStack = new Stack(10);

...

IStack stack = new SafeStack(20);
if (stack instanceof SafeStack myStack) {    // Local variable introduced.
  System.out.println(myStack.isFull());      // Shadows field reference.
}
myStack.push("Hello");                       // Field reference.
```

```
IStack stack = new SafeStack(
if (stack instanceof final SafeStack safestack) {  // (1) final pattern variable.
  safestack = new SafeStack(100);                  // (2) Compile-time error!
  System.out.println(safestack.isFull());
}
```

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack) {
  System.out.println(safestack.isFull());   // Pattern variable in scope.
} else {
  System.out.println(safestack.isEmpty());  // Compile-time error!
}
```

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) { // Logical complement operator (!)
  System.out.println(safestack.isFull());      // Compile-time error.
} else {
  System.out.println(safestack.isFull());      // Pattern variable in scope.
}
```

```
IStack stack = new SafeStack(20);
if (!(stack instanceof SafeStack safestack)) {
  System.out.println("No safestack here");
  return;                                     // (1) Does not complete normally.
} else {
  System.out.println(safestack.isFull());     // Pattern variable in scope.
}
System.out.println(safestack.isEmpty());      // Pattern variable still in scope.
```

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack && safestack.isFull()) {
  System.out.println("safestack is full");
  Object obj = safestack.pop();
}
```

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack || safestack.isFull()) { // Compile-time
                                                                     error.
  System.out.println("safestack is full");
  Object obj = safestack.pop();                               // Compile-time error.
}
```

```
IStack stack = new SafeStack(20);
if (stack instanceof SafeStack safestack &&  // Compile-time error!
    stack instanceof ISafeStack safestack) { // Duplicate variable safestack
  System.out.println(safestack);
}
```

```
IStack stack = new SafeStack(20);
boolean result = stack instanceof SafeStack safestack ? safestack.isEmpty()
                                                     : false;
```

```
// File: PolymorphRefs.java
interface IDrawable {                                                // (1)
  void draw();
}
//_______________________________________________________________________________
abstract class Shape implements IDrawable {
  abstract public void area();
}
//_______________________________________________________________________________
class Circle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Circle."); }
  @Override public void area() {
    System.out.println("Computing area of a Circle.");
  }
}
//_______________________________________________________________________________
class Rectangle extends Shape {
  @Override public void draw() { System.out.println("Drawing a Rectangle."); }
  @Override public void area() {
    System.out.println("Computing area of a Rectangle.");
  }
}
//_______________________________________________________________________________
class Square extends Rectangle {
  @Override public void draw() { System.out.println("Drawing a Square."); }
  @Override public void area() {
    System.out.println("Computing area of a Square.");
  }
}
//_______________________________________________________________________________
class Graph implements IDrawable {
  @Override public void draw() { System.out.println("Drawing a Graph."); }
}
//_______________________________________________________________________________
public class PolymorphRefs {
  public static void main(String[] args) {
    IDrawable[] drawables
        = {new Square(), new Circle(), new Rectangle(), new Graph()};     // (2)

    System.out.println("Draw drawables:");
    for (IDrawable drawable : drawables) {                           // (3)
      drawable.draw();
    }

    System.out.println("Only draw shapes:");
    for (IDrawable drawable : drawables) {                           // (4)
      if (drawable instanceof Shape) {                               // (5)
        drawable.draw();
      }
    }

    System.out.println("Only compute area of shapes:");
    for (IDrawable drawable : drawables) {                           // (6)
```

```
      if (drawable instanceof Shape shape) {                                      // (7)
        shape.area();                                                               // (8)
      }
    }
  }
}
```

```
Draw drawables:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Drawing a Graph.
Only draw shapes:
Drawing a Square.
Drawing a Circle.
Drawing a Rectangle.
Only compute area of shapes:
Computing area of a Square.
Computing area of a Circle.
Computing area of a Rectangle.
```

```
IDrawable d1 = new Rectangle();
System.out.println(d1 instanceof IDrawable); // true. Rectangle is an IDrawable.
System.out.println(d1 instanceof Shape);     // true. Rectangle is a Shape.
System.out.println(d1 instanceof Rectangle); // true. Rectangle is a Rectangle.
System.out.println(d1 instanceof Circle);    // false. Rectangle is not a Circle.
System.out.println(d1 instanceof Graph);     // false. Rectangle is not a Graph.
// System.out.println(d1 instanceof String); // Unrelated. Compile-time error.
```

```
IDrawable d2 = new Square();  // Subtype object denoted by supertype reference.
d2.area();        // Method not defined for type IDrawable. Compile-time error!
```

```
((Square) d2).area();           // Prints "Computing area of a Square."
((Shape) d2).area();            // Prints "Computing area of a Square."
```

```
IDrawable d3 = new Graph();        // No compile-time error.
((Shape) d3).area();               // Throws a ClassCastException!
```

```
if (d2 instanceof Shape shape) {           // true
  shape.area();                            // Prints "Computing area of a Square."
} else {
  System.out.println(d2.getClass().getName() + " is not a Shape." );
}

if (d3 instanceof Shape shape) {           // false
  shape.area();
} else {
  System.out.println(d3.getClass().getName() +
                     " is not a Shape." ); // Prints "Graph is not a Shape."
}
```

```
// Class declarations:
class Super {}
class Sub extends Super {}

// Reference declarations:
  Super x = null;
  Sub y = null;
```

```
class A { }
class B extends A { }
class C extends B { }
```

```
interface I{}
interface J{}
class C implements I {}
class D extends C implements J {}

public class RQ07A100 {
  public static void main(String[] args) {
    I x = new D();
    if (x instanceof I) System.out.print("I");
    if (x instanceof J) System.out.print("J");
    if (x instanceof C) System.out.print("C");
    if (x instanceof D) System.out.print("D");
    System.out.println();
  }
}
```

```
public class RQ800A20 {
  static void compute(int... ia) {                                // (1)
    System.out.print("|");
    for(int i : ia) {
      System.out.print(i + "|");
    }
    System.out.println();
  }
  static void compute(int[] ia1, int... ia2) {                    // (2)
    compute(ia1);
    compute(ia2);
  }
  static void compute(int[] ia1, int[]... ia2d) {                  // (3)
    for(int[] ia : ia2d) {
      compute(ia);
    }
  }
  public static void main(String[] args) {
    compute(new int[] {10, 11}, new int[] {12, 13, 14});          // (4)
    compute(15, 16);                                              // (5)
    compute(new int[] {17, 18}, new int[][] {{19}, {20}});        // (6)
    compute(null, new int[][] {{21}, {22}});                      // (7)
  }
}
```

```
public class RQ43 {
  public static void main(String[] args) {
    Object x = "acme";
    if (x instanceof String s && s.length() > 5) {
      x = s.equals("acme") ? "1" : "2";
    } else {
      x = s.equals("acme") ? "3" : "4";
    }
    System.out.println(x);
  }
}
```

```
Integer x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

```
var x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

```
var x = Integer.valueOf(42);
if (x instanceof String s) {
    System.out.print(s.toUpperCase());
}
```

```
Number x = Integer.valueOf(42);
if (x instanceof Integer s) {
    System.out.print(s.intValue());
}
```

```
Integer x = Integer.valueOf(42);
if (x instanceof Number s) {
    System.out.print(s.intValue());
}
```

```
class A           { int f() { return 0; } }
class B extends A { int f() { return 1; } }
class C extends B { int f() { return 2; } }

public class Polymorphism {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.f());
  }
}
```

```
class A {
  private int f() { return 0; }
  public int g() { return 3; }
}
class B extends A {
  private int f() { return 1; }
  public int g() { return f(); }
}
class C extends B {
  public int f() { return 2; }
}

public class Polymorphism2 {
  public static void main(String[] args) {
    A ref1 = new C();
    B ref2 = (B) ref1;
    System.out.println(ref2.g());
  }
}
```

```
public interface HeavenlyBody { String describe(); }

class Star {
  String starName;
  public String describe() { return "star " + starName; }
}

class Planet extends Star {
  String name;
  public String describe() {
    return "planet " + name + " orbiting star " + starName;
  }
}
```

```
public class MachineState {
  public static final int BUSY = 1;
  public static final int IDLE = 0;
  public static final int BLOCKED = -1;
}
```

```
access_modifier enum enum_type_name     // Enum header
{                                       // Enum body
  EC1, EC2, ..., ECk                     // Enum constants
}
```

```
public enum MachineState                  // Enum header
{                                         // Enum body
  BUSY, IDLE, BLOCKED                     // Enum constants

}
```

```
public enum MarchingOrders { LEFT, RIGHT }
public enum TrafficLightState { RED, YELLOW, GREEN; }
enum MealType { BREAKFAST, LUNCH, DINNER }
```

```
// File: MachineState.java
public enum MachineState { BUSY, IDLE, BLOCKED }
```

```
// File: Machine.java
public class Machine {

  private MachineState state;

  public void setState(MachineState state) { this.state = state; }
  public MachineState getState()           { return this.state; }
}
```

```
// File: MachineClient.java
public class MachineClient {
  public static void main(String[] args) {

    Machine machine = new Machine();
    machine.setState(MachineState.IDLE);           // (1) Passed as a value.
    // machine.setState(1);                         // (2) Compile error!
    MachineState state = machine.getState();        // (3) Declaring a reference.
    System.out.println(
        "Current machine state: " + state           // (4) Printing the enum name.
    );

    // MachineState newState = new MachineState();  // (5) Compile error!

    System.out.println("All machine states:");
    for (MachineState ms : MachineState.values()) { // (6) Traversing over enum
      System.out.println(ms + ":" + ms.ordinal());  //     contants.
    }

    System.out.println("Comparison:");
    MachineState state1 = MachineState.BUSY;
    MachineState state2 = state1;
    MachineState state3 = MachineState.BLOCKED;

    System.out.println(state1 + " == " + state2 + ": " +
                       (state1 == state2));                          // (7)
    System.out.println(state1 + " is equal to " + state2 + ": " +
                       (state1.equals(state2)));                     // (8)
    System.out.println(state1 + " is less than " + state3 + ": " +
                       (state1.compareTo(state3) < 0));              // (9)
  }
}
```

```
Current machine state: IDLE
All machine states:
BUSY:0
IDLE:1
BLOCKED:2
Comparison:
BUSY == BUSY: true
BUSY is equal to BUSY: true
BUSY is less than BLOCKED: true
```

```
// File: Meal.java
public enum Meal {
  BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);            // (1)

  // Non-zero argument constructor                            (2)
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Fields for the meal time:                                (3)
  private int hh;
  private int mm;

  // Instance methods:                                        (4)
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                           // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}
```

```
// File: MealAdministrator.java
public class MealAdministrator {
  public static void main(String[] args) {

    System.out.printf(                                       // (5)
        "Please note that no eggs will be served at %s, %s.%n",
        Meal.BREAKFAST, Meal.BREAKFAST.getTimeString()
    );

    System.out.println("Meal times are as follows:");
    Meal[] meals = Meal.values();                            // (6)
    for (Meal meal : meals) {                                // (7)
      System.out.printf("%s served at %s%n", meal, meal.getTimeString());
    }

    Meal formalDinner = Meal.valueOf("DINNER");              // (8)
    System.out.printf("Formal dress is required for %s at %s.%n",
        formalDinner, formalDinner.getTimeString()
    );
  }
}
```

Please note that no eggs will be served at BREAKFAST, 07:30.
Meal times are as follows:
BREAKFAST served at 07:30
LUNCH served at 12:15
DINNER served at 19:45
Formal dress is required for DINNER at 19:45.

```
static EnumTypeName[] values()
static EnumTypeName valueOf(String name)
```

```
final boolean equals(Object other)
```

```
BREAKFAST(7,30) {                    // (1) Start of constant-specific class body
  @Override
  public double mealPrice(Day day) {  // (2) Overriding abstract method
    ...
  }
  @Override
  public String toString() {          // (3) Overriding method from the Enum class
    ...
  }
}                                     // (4) End of constant-specific class body
```

```
// File: Day.java
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

```
// File: Meal.java
public enum Meal {
  // Each enum constant defines a constant-specific class body
  BREAKFAST(7,30) {                                                   // (1)
    @Override
    public double mealPrice(Day day) {                                // (2)
      double breakfastPrice = 10.50;
      if (day.equals(Day.SATURDAY) || day == Day.SUNDAY)
        breakfastPrice *= 1.5;
      return breakfastPrice;
    }
    @Override
    public String toString() {                                        // (3)
      return "Breakfast";
    }
  },                                                                  // (4)
  LUNCH(12,15) {
    @Override
    public double mealPrice(Day day) {                                // (5)
      double lunchPrice = 20.50;
      switch (day) {
        case SATURDAY: case SUNDAY:
          lunchPrice *= 2.0;
      }
      return lunchPrice;
    }
    @Override
    public String toString() {
      return "Lunch";
    }
  },
  DINNER(19,45) {
    @Override
    public double mealPrice(Day day) {                                 // (6)
      double dinnerPrice = 25.50;
      if (day.compareTo(Day.SATURDAY) >= 0 && day.compareTo(Day.SUNDAY) <= 0)
        dinnerPrice *= 2.5;
      return dinnerPrice;
    }
  };

  // Abstract method implemented in constant-specific class bodies.
  abstract double mealPrice(Day day);                                  // (7)

  // Enum constructor:
  Meal(int hh, int mm) {
    this.hh = hh;
    this.mm = mm;
  }

  // Instance fields: Time for the meal.
  private int hh;
  private int mm;

  // Instance methods:
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
  public String getTimeString() {                               // "hh:mm"
    return String.format("%02d:%02d", this.hh, this.mm);
  }
}

// File: MealPrices.java
public class MealPrices {

  public static void main(String[] args) {                             // (8)
    System.out.printf(
        "Please note that %s, %s, on %s costs $%.2f.%n",
        Meal.BREAKFAST.name(),                                         // (9)
        Meal.BREAKFAST.getTimeString(),
        Day.MONDAY,
        Meal.BREAKFAST.mealPrice(Day.MONDAY)                           // (10)
    );

    System.out.println("Meal prices on " + Day.SATURDAY + " are as follows:");
    Meal[] meals = Meal.values();
    for (Meal meal : meals) {
      System.out.printf(
          "%s costs $%.2f.%n", meal, meal.mealPrice(Day.SATURDAY)       // (11)
      );
    }
  }
}
```

Please note that BREAKFAST, 07:30, on MONDAY costs $10.50.

```
Meal prices on SATURDAY are as follows:
Breakfast costs $15.75.
Lunch costs $41.00.
DINNER costs $63.75.
```

```
Day day = Day.MONDAY;             // See Example 5.26, p. 295, for enum type Day.
String typeOfDay = switch (day) {                                          // (1)
  case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";          // (2)
  case SATURDAY, SUNDAY                             -> "Weekend";          // (3)
};
```

```
typeOfDay = switch (day) {                                                    // (4)
  case SATURDAY, SUNDAY -> "Weekend";                                         // (5)
  default -> "Weekday";                                                       // (6)
}
```

```
public class MealPrices {
  public enum Day { /* ... */ }                         // Static member

  public static enum Meal { /* ... */ }                 // Static member

  public static void main(String[] args) { /* ... */ }   // Static method
}
```

```
public interface ITimeInfo {
  public int getHour();
  public int getMins();
}

public enum Meal implements ITimeInfo {
  // ...
  @Override public int getHour() { return this.hh; }
  @Override public int getMins() { return this.mm; }
  // ...
}
```

```
package record.basics;
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
```

```
package record.basics;
import java.time.Year;
import java.util.Objects;

/** A class that represents a CD. */
public class CD_v1 {

  // Instance fields:
  private final String artist;      // Name of the artist.
  private final String title;       // Title of the CD.
  private final int    noOfTracks;  // Number of tracks on the CD.
  private final Year   year;        // Year the CD was released.
  private final Genre  genre;       // Music genre of the CD.

  // Non-zero argument constructor:
  public CD_v1(String artist, String title, int noOfTracks,
               Year year, Genre genre) {
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
    this.genre      = genre;
  }
  // Get methods:
  public String getArtist()     { return this.artist; }
  public String getTitle()      { return this.title; }
  public int    getNoOfTracks() { return this.noOfTracks; }
  public Year   getYear()       { return this.year; }
  public Genre  getGenre()      { return this.genre; }

  // Overridden methods from the Object class:
  @Override public String toString() {
    return String.format("<%s, \"%s\", %d, %s, %s>",
        this.artist, this.title, this.noOfTracks, this.year, this.genre);
  }

  @Override public boolean equals(Object obj) {
    return (this == obj)
        || (obj instanceof CD_v1 other
            && this.artist.equals(other.artist)
            && this.title.equals(other.title)
            && this.noOfTracks == other.noOfTracks
            && this.year == other.year
            && this.genre == other.genre);
  }

  @Override public int hashCode() {
    return Objects.hash(this.artist, this.title, this.noOfTracks,
                        this.year, this.genre);
  }
}
```

```
access_modifier record record_name(component_list)       // Record header
{ }                                                      // Empty record body
```

```
package record.basics;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,         // (1)
                 Year year, Genre genre) { /* Empty body */ }
```

```
package record.basics;
import java.time.Year;

public class DataUser {
  public static void main(String[] args) {
    // Some ready-made CDs:                                               (2)
    CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
    CD cd1 = new CD("Jaav",      "Java Jam",        6, Year.of(2017), Genre.JAZZ);
    CD cd2 = new CD("Funkies",   "Lambda Dancing", 10, Year.of(2018), Genre.POP);
    CD cd3 = new CD("Genericos", "Keep on Erasing", 8, Year.of(2018), Genre.JAZZ);
    CD cd4 = new CD("Genericos", "Hot Generics",   10, Year.of(2018), Genre.JAZZ);

    // An array of CDs.
    CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};

    System.out.println("     Artist    Title           No. Year Genre");
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
          cd.artist(), cd.title(), cd.noOfTracks(),                   // (3)
          cd.year(), cd.genre());
      System.out.printf("cd%d: %s%n", i, cdToString);
    }

    System.out.println();
    System.out.println(cd0.toString());                                // (4)

    CD cdX = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
    System.out.println("cd0.equals(cdX): " + cd0.equals(cdX));         // (5)
  }
}
```

```
     Artist    Title           No. Year Genre
cd0: Jaav      Java Jive       8   2017 POP
cd1: Jaav      Java Jam        6   2017 JAZZ
cd2: Funkies   Lambda Dancing  10  2018 POP
```

```
cd3: Genericos Keep on Erasing 8   2018 JAZZ
cd4: Genericos Hot Generics    10  2018 JAZZ

cd0: CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
cd0.equals(cdX): true
```

```
CD cd0 = new CD("Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
...
```

```
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

access_modifier `record` *record_name*(*component_list*) *optional_implements_clause* {

  optional_constructor_declarations

  optional_member_declarations

}

```
package record.constructors.canonical;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,          // (1)
                 Year year, Genre genre) {

  // Normal canonical record constructor                              // (2)
  public CD(String artist, String title, int noOfTracks, Year year, Genre genre) {
    // Sanitize the parameter values:                                    (3)
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;
    // Initialize all component fields:                                       (4)
    this.artist     = artist;
    this.title      = title;
    this.noOfTracks = noOfTracks;
    this.year       = year;
    this.genre      = genre;
  }
}
```

```
package record.constructors.compact;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,             // (1)
                 Year year, Genre genre) {

  // Compact canonical record constructor                                    (2)
  public CD {
    // Sanitize the values passed to the constructor:                        (3)
    artist = artist.strip();
    title = title.strip();
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;

    // Cannot explicitly assign to component fields:                         (4)
    // this.artist = artist;                    // Compile-time error!
  }

  // A non-canonical record constructor                                      (5)
  public CD() {
    this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (6)
  }
}
```

```
System.out.println(new CD());         // Calls the non-canonical constructor
```

```
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
```

```
package record.customize;
import java.time.Year;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,             // (1)
                 Year year, Genre genre) {

  // Compact canonical record constructor                                    (2)
  public CD {
    // Sanitize the values passed to the constructor:                        (3)
    artist = artist.strip();
    title = title.strip();                                                // (4a)
//  title = title.strip().toUpperCase();                                  // (4b)
    noOfTracks = noOfTracks < 0 ? 0 : noOfTracks;
    year =  year.compareTo(Year.of(2022)) > 0? Year.of(2022) : year;
    genre = genre == null ? Genre.OTHER : genre;

// Cannot explicitly assign to component fields:                             (5)
// this.artist = artist;                                  // Compile-time error!
  }

  // A non-canonical record constructor                                      (6)
  public CD() {
    this(" Anonymous  ", "  No title  ", 0, Year.of(2022), Genre.OTHER);  // (7)
  }

  //  New instance methods:                                                  (8)
  public boolean isPop()   { return this.genre == Genre.POP; }
  public boolean isJazz()  { return this.genre == Genre.JAZZ; }
  public boolean isOther() { return this.genre == Genre.OTHER; }

  // Customize a get method:                                                 (9)
  @Override public String title() {
     return this.title.toUpperCase();
  }

  // Static fields with some ready-made CDs:                                 (10)
  public final static CD cd0
      = new CD("   Jaav",      "Java Jive",       8, Year.of(2017), Genre.POP);
  public final static CD cd2
      = new CD("Funkies ",   " Lambda Dancing  ", 10, Year.of(2024), null);
  public final static CD cd4
      = new CD("Genericos", "Hot Generics",   -5, Year.of(2018), Genre.JAZZ);

  // Declare a nested type:
  public enum Genre {POP, JAZZ, OTHER}                                   //  (11)
}
```

```
package record.customize;

public class DataUser2 {
  public static void main(String[] args) {
    CD[] cdArray = {CD.cd0, CD.cd2, CD.cd4, new CD()};                    // (12)
    for(int i = 0; i < cdArray.length; ++i) {
      CD cd = cdArray[i];
      if (cd.isOther()) {                                                 // (13)
        System.out.println(cd.toString());                                // (14)
      }
    }
    System.out.println("Title: " + cdArray[1].title());                   // (15)
  }
}
```

```
CD[artist=Funkies, title=Lambda Dancing, noOfTracks=10, year=2022, genre=OTHER]
CD[artist=Anonymous, title=No title, noOfTracks=0, year=2022, genre=OTHER]
Title: LAMBDA DANCING
```

```
public record CD(String artist, String title, int noOfTracks,          // (1)
    Year year, Genre genre) implements Serializable and Comparable<CD> {

  @Override public int compareTo(CD other) { /* See Example 16.1, p. 883. */ }

  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

```
record Container<T>(T item) { /* Empty body */ }
```

```
Container<String> p0 = new Container<>("Hi");
Container<Integer> p1 = new Container<>(Integer.valueOf(10));
```

```
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {/*(1)*/}
public non-sealed class PrintedBook extends Book {/*...*/}

public final class Ebook extends Book {/*...*/}

public final class Audiobook extends Book {/*...*/}
```

```
class EComicBook extends Ebook {}   // Compile-time error!
class Hardcover extends PrintedBook {}
class Paperback extends PrintedBook {}
```

```
public sealed class PrintedBook extends Book permits Hardcover, Paperback {}
final class Hardcover extends PrintedBook {}
final class Paperback extends PrintedBook {}
```

```
package signedandsealed;

public abstract sealed class Book permits PrintedBook, Ebook, Audiobook  { // (1)
  private String isbn;
  protected Book(String isbn) {
    this.isbn = isbn;
  }
  public String getIsbn() { return this.isbn; }
}
```

```
package signedandsealed;

public non-sealed class PrintedBook extends Book {                         // (2)
  private int pageCount;
  protected PrintedBook(String isbn, int pageCount) {
    super(isbn);
    this.pageCount = pageCount;
  }
  public int getPageCount() { return this.pageCount; }
}
```

```
package signedandsealed;

public final class Ebook extends Book {                                     // (3)
  private String format;
  public Ebook(String isbn, String format) {
    super(isbn);
    this.format = format;
  }
  public String getFormat() { return this.format; }
}
```

```
package signedandsealed;

public final class Audiobook extends Book {                              // (4)
  private String narrator;
  private double length;
  public Audiobook(String isbn, String narrator, double length) {
    super(isbn);
    this.narrator = narrator;
    this.length = length;
  }
  public String getNarrator() { return this.narrator; }
  public double getLength() { return this.length; }
}
```

```
public sealed interface Subscribable permits Ebook, Audiobook, VIPSubcribable {}
public final class Ebook extends Book implements Subscribable {}
public final class Audiobook extends Book implements Subscribable {}
public non-sealed interface VIPSubcribable extends Subscribable {}
public abstract sealed class Book permits PrintedBook, Ebook, Audiobook {}
public non-sealed class PrintedBook extends Book {}
```

```
sealed interface MediaStorage permits CD, DVD {}       // (1) Sealed interface
enum CD implements MediaStorage {CD_ROM, CD_R, CD_W}   // (2) Implicitly final
enum DVD implements MediaStorage {DVD_R {}, DVD_RW}    // (3) Implicitly sealed
```

```
sealed interface MediaStorage permits CD, DVD, HardDisk {}// (1a) Sealed interface
record HardDisk(double capacity) implements MediaStorage {}// (4) Implicitly final
```

```
// File: Book.java (compilation unit)
public abstract sealed class Book {}        // Permitted subclasses are derived.
non-sealed class PrintedBook extends Book {}

sealed interface Subscribable {}            // Permitted subtypes are derived.
final class Ebook extends Book implements Subscribable {}
final class Audiobook extends Book implements Subscribable {}
non-sealed interface VIPSubcribable extends Subscribable {}
```

```
package signedandsealed;

public class BookAdmin {
  public static void main(String[] args) {
    // Create some books:                                             // (1)
    PrintedBook pBook = new PrintedBook("888-222", 340);
    Ebook eBook = new Ebook("999-777", "epub");
    Audiobook aBook = new Audiobook("333-555", "Fry", 300.0);

    // Create a book array:                                           // (2)
    Book[] books = {pBook, eBook, aBook};

    // Process the books:                                             // (3)
    for (Book book : books) {
      if (book instanceof PrintedBook pb) {                           // (4)
        System.out.printf("Printed book: %s, %d%n",
                          pb.getIsbn(), pb.getPageCount());
      } else if (book instanceof Ebook eb) {
        System.out.printf("Ebook: %s, %s%n", eb.getIsbn(), eb.getFormat());
      } else if (book instanceof Audiobook ab) {
        System.out.printf("Audiobook: %s, %s, %.1f%n",
                          ab.getIsbn(), ab.getNarrator(), ab.getLength());
      }
    }
  }
}
```

```
Printed book: 888-222, 340
Ebook: 999-777, epub
Audiobook: 333-555, Fry, 300.0
```

```
public enum Drill {
  ATTENTION("Attention!"), EYES_RIGHT("Eyes right!"),
  EYES_LEFT("Eyes left!"), AT_EASE("At ease!");

  private String command;

  Drill(String command) {
    this.command = command;
  }
  public static void main(String[] args) {
    System.out.println(ATTENTION);                // (1)
    System.out.println(AT_EASE);                  // (2)
  }
}
```

```
import java.util.Arrays;

public enum Priority {
  ONE(1) { public String toString() { return "LOW"; } },        // (1)
  TWO(2),
  THREE(3) { public String toString() { return "NORMAL"; } },   // (2)
  FOUR(4),
  FIVE(5) { public String toString() { return "HIGH"; } };      // (3)

  private int pValue;

  Priority(int pValue) {
    this.pValue = pValue;
  }

  public static void main(String[] args) {
    System.out.println(Arrays.toString(Priority.values()));
  }
}
```

```
public enum TrafficLight {
  RED("Stop"), YELLOW("Caution"), GREEN("Go");

  private String action;

  TrafficLight(String action) {
    this.action = action;
  }

  public static void main(String[] args) {
    TrafficLight green = new TrafficLight("Go");
    System.out.println(GREEN.equals(green));
  }
}
```

```
public class RQ54 {
  public static void main(String[] args) {
    Product p = new Product(101, "Tea", 1.99);
    System.out.println(p);
  }
}
```

```
Product[id=101, name=TEA, price=1.99]
```

```
public record Product(int id, String name, double price) {
  public Product() {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

```
public record Product(int id, String name, double price) {
  public Product(int newId, String newName, double newPrice) {
    id = newId;
    name = newName.toUpperCase();
    price = newPrice;
  }
}
```

```
public record Product() {
  public Product(int id, String name, double price) {
    this.id = id;
    this.name = name.toUpperCase();
    this.price = price;
  }
}
```

```
public record Product(int id, String name, double price) {
  public Product {
    name = name.toUpperCase();
  }
}
```

```
public record Product(int id, String name, double price) {
  public String toString(){
    return "Product[id="+id+", name="+name.toUpperCase()+",
            price="+price+"]";
  }
}
```

```
public record Product(int id, String name, double price) {
  public String name() {
    return name.toUpperCase();
  }
  public String toString(){
    return "Product[id="+id+", name="+name+", price="+price+"]";
  }
}
```

```
public record Employee(String name, double salary) { }
public class RQ55 {
  public static void main(String[] args) {
    Employee e1 = new Employee("Bob", 1234);
    Employee e2 = new Employee("Bob", 1234);
    System.out.print(e1.equals(e2));
    System.out.print(e1 == e2);
  }
}
```

```
public sealed interface X permits Y, Z { }
public final class Y implements X { }
public sealed interface Z extends X { }
public final class Y implements X { }
public interface Z extends X { }
public non-sealed class Y implements X, Z { }
public sealed interface Z extends X permits Y { }
public class Y implements X, Z { }
public non-sealed interface Z extends X { }
```

`package` *fully_qualified_package_name*;

```
// File name: Clown.java                      // This file has 2 type declarations
package wizard.pandorasbox;                   // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing specific class

public class Clown implements Magic { /* ... */ }

interface Magic { /* ... */ }
```

```
// File name: LovePotion.java
package wizard.pandorasbox;                  // Package declaration

public class LovePotion { /* ... */ }
```

```
// File name: Ailment.java
package wizard.pandorasbox.artifacts;        // Package declaration

public class Ailment { /* ... */ }
```

```
// File name: Baldness.java                 // This file has 2 type declarations
package wizard.spells;                      // Package declaration

import wizard.pandorasbox.*;                // (1) Type-import-on-demand
import wizard.pandorasbox.artifacts.*;      // (2) Import from subpackage

public class Baldness extends Ailment {     // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;     // (3) Fully qualified class name
  LovePotion tlcTwo;                        // Class in same package
  // ...
}

class LovePotion { /* ... */ }
```

```
import fully_qualified_type_name;
import wizard.pandorasbox.Clown;
import fully_qualified_package_name.*;
import wizard.pandorasbox.*;
import java.awt.*;           // imports all reference types from java.awt
import java.awt.*;              // imports all type names from java.awt
import java.util.*;             // imports all type names from java.util
import java.awt.*;           // imports all type names from java.awt
import java.util.*;          // imports all type names from java.util
import java.awt.List;        // imports the type List from java.awt explicitly
import static java.lang.Math.*;
```

```
double hypotenuse = hypot(x, y);   // (3') Type name can now be omitted.
```

```
import static java.lang.Math.PI;           // (1) Static field
import static java.lang.Math.sqrt;         // (2) Static method
// Only specified static members are imported.

public class Calculate3 {
  public static void main(String[] args) {
    double x = 3.0, y = 4.0;
    double squareroot = sqrt(y);            // Simple name of static method
    double hypotenuse = Math.hypot(x, y);  // (3) Requires type name
    double area = PI * y * y;               // Simple name of static field
    System.out.printf("Square root: %.2f, hypotenuse: %.2f, area: %.2f%n",
                      squareroot, hypotenuse, area);
  }
}
```

```
Square root: 2.00, hypotenuse: 5.00, area: 50.27
```

```
public class MyFactory implements mypkg.IMachineState {
 // ...
}
```

```
package mypkg;

public interface IMachineState {
  // Fields are public, static, and final.
  int BUSY = 1;
  int IDLE = 0;
  int BLOCKED = -1;
}
```

```
import static mypkg.IMachineState.*;    // (1) Static import interface constants

public class MyFactory {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED }; // (2) Access by simple name
    for (int s : states)
      System.out.print(s + " ");
  }
}
```

```
package mypkg;

public enum State { BUSY, IDLE, BLOCKED }
```

```
// File: Factory.java (in unnamed package)
import mypkg.State;                  // (1) Single type import

import static mypkg.State.*;         // (2) Static import on demand
import static java.lang.System.out;  // (3) Single static import

public class Factory {
  public static void main(String[] args) {
    State[] states = {               // (4) Using type import implied by (1)
        IDLE, BUSY, IDLE, BLOCKED    // (5) Using static import implied by (2)
    };
    for (State s : states)           // (6) Using type import implied by (1)
      out.print(s + " ");            // (7) Using static import implied by (3)
  }
}
```

```
import static java.lang.System.out;       // (1) Static import

public class ShadowImport {

  public static void main(String[] args) {
    out.println("Calling println() in java.lang.System.out");
    ShadowImport sbi = new ShadowImport();
    writeInfo(sbi);
  }
  // Parameter out shadows java.lang.System.out:
  public static void writeInfo(ShadowImport out) {
    out.println("Calling println() in the parameter out");
    System.out.println("Calling println() in java.lang.System.out"); // Qualify
  }

  public void println(String msg) {
    out.println(msg + " of type ShadowImport");
  }
}
```

```
Calling println() in java.lang.System.out
Calling println() in the parameter out of type ShadowImport
Calling println() in java.lang.System.out
```

```
import static java.lang.Integer.MAX_VALUE;
import static java.lang.Double.MAX_VALUE;

public class StaticFieldConflict {
  public static void main(String[] args) {
    System.out.println(MAX_VALUE);          // (1) Ambiguous! Compile-time error!
    System.out.println(Integer.MAX_VALUE);  // OK
    System.out.println(Double.MAX_VALUE);   // OK
  }
}
```

```
package mypkg;

public class Auxiliary {
  public static int binarySearch(int[] a, int key) { // Same in java.util.Arrays
    // Implementation is omitted.
    return -1;
  }
}
```

```
// File: MultipleStaticImport.java (in unnamed package)
import static java.util.Collections.binarySearch;  //    2 overloaded methods
import static java.util.Arrays.binarySearch;       // + 18 overloaded methods
import static mypkg.Auxiliary.binarySearch; // (1) Causes signature conflict

public class MultipleStaticImport {
  public static void main(String[] args) {
    int index = binarySearch(new int[] {10, 50, 100}, 50); // (2) Ambiguous!
    System.out.println(index);
  }

//public static int binarySearch(int[] a, int key) {       // (3)
//  return -1;
//}
}
```

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
```

```
>javac -d /pgjc/work Clown.java LovePotion.java Ailment.java Baldness.java
```

```
>javac -d ../myapp Clown.java LovePotion.java Ailment.java Baldness.java
```

```
>java wizard.pandorasbox.Clown
```

```
>javac -d ../bin A.java
A.java:3: cannot find symbol
symbol  : class B
location: class pkg.A
public class A { B b; }
                 ^
1 error
```

```
>javac -classpath /top/bin -d ../bin A.java
```

```
>javac -classpath /top/bin/pkg -d ../bin A.java
```

```
>java -classpath /pgjc/work/wizard pandorasbox.Clown
```

```
>java -classpath /pgjc/work wizard.pandorasbox.Clown
```

```
-classpath /pgjc/work:/top/bin:.
```

```
-classpath /ext/lib1:/ext/lib2
-classpath /ext/lib2:/ext/lib1
```

```
>java -classpath . wizard.pandorasbox.Clown
```

```
>javac -classpath ../bin -d ../bin A.java
```

```
package net.alphabet;
import java.util.ArrayList;
public class A {}
class B {}
```

```
package app;
public class Window {
  final static String frame = "Top-frame";
}
```

```
package app;
// (1) INSERT IMPORT STATEMENT HERE
public class Canvas {
  private String str = frame;
}
```

```
package mainpkg.subpkg1;
public class Window {}
```

```
package mainpkg.subpkg2;
public class Window {}
```

```
package mainpkg;
// (1) INSERT IMPORT STATEMENTS HERE
public class Screen {
  private Window win;
}
```

```
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.Window;
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.*;
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.*;
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.Window;
```

```
// (1) INSERT ONE IMPORT STATEMENT HERE
public class RQ700A20 {
  public static void main(String[] args) {
    System.out.println(sqrt(49));
  }
}
```

```
/top
  |--- wrk
        |--- pkg
              |--- A.java
              |--- B.java
```

```
Current directory: /top/wrk
Command: javac -cp .:pkg A.java
```

```
Current directory: /top/wrk
Command: javac -cp . pkg/A.java
```

```
Current directory: /top/wrk
Command: javac -cp pkg A.java
```

```
Current directory: /top/wrk
Command: javac -cp .:pkg pkg/A.java
```

```
Current directory: /top/wrk/pkg
Command: javac A.java
```

```
Current directory: /top/wrk/pkg
Command: javac -cp . A.java
```

```
package a.b;
public class X {
  public static int y = 100;
}
```

```
package a.b.c;
// (1) INSERT IMPORT STATEMENT HERE
public class Z {
  public void xyz() {
        int v = y;
      }
    }
```

```
package life.animals;
public class Cat { }
```

```
package life.animals;
public class Cow { }
```

```
package life.animals;
public class Dog { }
```

```
package habitat;
// (1) INSERT IMPORT STATEMENTS HERE
public class Farm {
  private Cat cat;
  private Cow cow;
}
```

```
import static life.animals.Cat;
import static life.animals.Cow;
```

```
// File: Clown.java
package wizard.pandorasbox;                  // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing class Ailment

public class Clown implements Magic {        // (1)
  LovePotion tlc;                            // Class in same package
  Ailment problem;                           // Simple class name
  Clown() {
    tlc = new LovePotion("passion");
    problem = new Ailment("flu");            // Simple class name
  }
  @Override public void levitate()  {        // (2)
    System.out.println("Levitating");
  }
  public void mixPotion()   { System.out.println("Mixing " + tlc); }
  public void healAilment() { System.out.println("Healing " + problem); }

  public static void main(String[] args) {
    Clown joker = new Clown();
    joker.levitate();
    joker.mixPotion();
    joker.healAilment();
  }
}
```

```
interface Magic { void levitate(); }          // (3)
```

```
// File: LovePotion.java
package wizard.pandorasbox;                   // Package declaration

public class LovePotion {                     // (4) Accessible outside package
  String potionName;
  public LovePotion(String name) { potionName = name; }
  public String toString()       { return potionName; }
}
```

```
// File: Ailment.java
package wizard.pandorasbox.artifacts;        // Package declaration

public class Ailment {                       // Accessible outside package
  String ailmentName;
  public Ailment(String name) { ailmentName = name; }
  public String toString() { return ailmentName; }
}
```

```
// File: Baldness.java
package wizard.spells;                        // Package declaration

import wizard.pandorasbox.*;                  // Redundant
import wizard.pandorasbox.artifacts.*;        // Import of subpackage

public class Baldness extends Ailment {       // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;       // Fully qualified name
  LovePotion tlcTwo;                          // Class in same package
  Baldness(String name) {
    super(name);
    tlcOne = new wizard.pandorasbox.          // Fully qualified name
                 LovePotion("romance");
    tlcTwo = new LovePotion();                // Class in same package
  }
}

class LovePotion /* implements Magic */ {     // (5) Magic is not accessible
  // @Override public void levitate() {}      // (6) Cannot override method
}
```

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
>java wizard.pandorasbox.Clown
Levitating
Mixing passion
Healing flu
```

```
// File: Superclass1.java
package pkg1;

import static java.lang.System.out;

public class Superclass1 {                            // (1)

  // Instance fields with different accessibility:
  public    int     pubInt   = 2017;
  protected String  proStr   = "SuperDude";
            boolean pgkBool  = true;
  private   long    privLong = 0x7777;

  public static void printState1() {                  // (2)
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);
  }
}

// Client 1
class Subclass1 extends Superclass1 {                 // (3)
  public static void printState1() {
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}

// Client 2
class NonSubclass1 {                                  // (4)
  public static void printState1() {
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}
```

```
// File: Subclass2.java
package pkg2;
import pkg1.Superclass1;

import static java.lang.System.out;
```

```
// Client 3
public class Subclass2 extends Superclass1 {     // (5)

  public static void printState1() {             // (6)
    Superclass1 obj1 = new Superclass1();        // Object of Superclass1
    out.println(obj1.pubInt);
    out.println(obj1.proStr);   // (7) Compile-time error! Protected access.
    out.println(obj1.pgkBool);  // Compile-time error! Package access.
    out.println(obj1.privLong); // Compile-time error! Private access.
  }

  public static void printState2() {             // (8)
    Subclass2 obj2 = new Subclass2();            // (9) Object of Subclass2
    out.println(obj2.pubInt);
    out.println(obj2.proStr);   // (10) OK! Protected access.
    out.println(obj2.pgkBool);  // Compile-time error! Package access.
    out.println(obj2.privLong); // Compile-time error! Private access.
  }
}

// Client 4
class NonSubclass2 {                             // (11)
  public static void printState1() {
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);   // Compile-time error! Protected access.
    out.println(obj1.pgkBool);  // Compile-time error! Package access.
    out.println(obj1.privLong); // Compile-time error! Private access.
  }
}
```

```
class SuperClass {
  int instanceVarInSuper;
  static int staticVarInSuper;
  void instanceMethodInSuper()      { /* ... */ }
  static void staticMethodInSuper() { /* ... */ }
  // ...
}

class MyClass extends SuperClass {
  int instanceVar;
  static int staticVar;

  void instanceMethod()      { /* ... */ }
  static void staticMethod() { /* ... */ }
  // ...
}
```

```
class Light {
  // Instance variables:
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  public static Light duplicateLight(Light oldLight) {     // (1)
    Light newLight = new Light();
    newLight.noOfWatts = oldLight.noOfWatts;               // (2)
    newLight.indicator = oldLight.indicator;               // (3)
    newLight.location  = oldLight.location;                // (4)
    return newLight;
  }
}
```

```
import java.util.Arrays;

public final class WeeklyStats {             // (1) Class is final.

  private final String description;          // (2) Immutable string
  private final int weekNumber;              // (3) Immutable primitive value
  private final int[] stats;                 // (4) Mutable int array
  public WeeklyStats(String description, int weekNumber, int[] stats) {   // (5)
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber);
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                          Arrays.toString(stats));
    }
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length);   // Create a private copy.
  }

  public int getWeekNumber() {                // (6) Returns immutable primitive.
    return weekNumber;
  }

  public String getDescription() {            // (7) Returns immutable string.
    return description;
  }

  public int getDayStats(int dayNumber) {     // (8) Returns stats for given day.
    return (0 <= dayNumber && dayNumber < 7) ? stats[dayNumber] : -1;
  }

  public int[] getStats() {                   // (9) Returns a copy of the stats.
    return Arrays.copyOf(this.stats, this.stats.length);
  }

  @Override
  public String toString() {
    return description + "(week " + weekNumber + "):" + Arrays.toString(stats);
  }
}
```

```
public class StatsClient {
  public static void main(String[] args) {
    WeeklyStats ws1
        = new WeeklyStats("Appointments", 45, new int[] {5, 3, 8, 10, 7, 8, 9});
    System.out.println(ws1);
    WeeklyStats ws2
        = new WeeklyStats("E-mails", 47, new int[] {10, 5, 20, 7});
    System.out.println(ws2);
  }
}
```

```
Appointments(week 45):[5, 3, 8, 10, 7, 8, 9]
Exception in thread "main" java.lang.IllegalArgumentException: Stats not for whole
week: [10, 5, 20, 7]
     at WeeklyStats.<init>(WeeklyStats.java:14)
     at StatsClient.main(StatsClient.java:7)
```

```
public class WeeklyStatsV2 {                    // (1) Class is not final.
  ...
  private WeeklyStatsV2(String description,
      int weekNumber, int[] stats) {            // (5a) Private constructor
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length); // Create a private copy.
  }

  // (5b) Static factory method to construct objects.
  public static WeeklyStatsV2 getNewWeeklyStats(String description,
                                               int weekNumber, int[] stats) {
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber);
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                         Arrays.toString(stats));
    }
    return new WeeklyStatsV2(description, weekNumber, stats);
  }
  ...
}
```

```
package com.corporation.project;

public class MyClass {
            int i;
  public    int j;
  protected int k;
  private   int l;
}
```

```
public class Dog {
  String name;
  public Dog(String name) {
    this.name = name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}
```

```
public class Average1 {

  public static void main(String[] args) {
    printAverage(100, 20);                                       // (1)
    System.out.println("Exit main().");                          // (2)
  }

  public static void printAverage(int totalSum, int totalCount) {
    int average = computeAverage(totalSum, totalCount);          // (3)
    System.out.println("Average = " +                            // (4)
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");                  // (5)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                    // (6)
    return sum/count;                                            // (7)
  }
}
```

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

```
printAverage(100, 20);                                      // (1)
```

```
    printAverage(100, 0);                                      // (1)
```

```
Computing average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

```
>java -Djava.compiler=NONE Average1
```

```
java.lang.ClassNotFoundException
```

java.io.FileNotFoundException

java.io.NotSerializableException

java.lang.ArithmeticException

```
java.lang.ArrayIndexOutOfBoundsException
```

java.lang.ArrayStoreException

java.lang.IllegalArgumentException

java.lang.NumberFormatException

java.lang.NullPointerException

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.toLowerCase()" because "msg" is null
    at StringMethods.main(StringMethods.java:162)
```

```
java.lang.UnsupportedOperationException
```

`java.time.format.DateTimeParseException` java.util.MissingResourceException

```
public class EvacuateException extends Exception {
  // Fields
  private Date date;
  private Zone zone;
  private TransportMode transport;

  // Constructor
  public EvacuateException(Date d, Zone z, TransportMode t) {
    // Call the constructor of the superclass, usually passing a detail message.
    super("Evacuation of zone " + z);
    // ...
  }
  // Methods
  // ...
}
```

```
try {                                          // try block
  statements
} catch (exception_type₁ parameter₁) {         // uni-catch clause
  statements
}
...
  catch (exception_typeₙ parameterₙ) {         // uni-catch clause
  statements
} finally {                                    // finally clause
  statements
}
```

```
public class Average2 {

  public static void main(String[] args) {
    printAverage(100, 20);                                      // (1)
    System.out.println("Exit main().");                         // (2)
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                       // (3)
      int average = computeAverage(totalSum, totalCount);       // (4)
      System.out.println("Average = " +                         // (5)
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                          // (6)
      ae.printStackTrace();                                     // (7)
      System.out.println("Exception handled in printAverage()."); // (8)
    }
    System.out.println("Exit printAverage().");                 // (9)
  }
  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                        // (10)
    return sum/count;                                                // (11)
  }
}
```

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:23)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

```
public class Average3 {

  public static void main(String[] args) {
    try {                                                        // (1)
      printAverage(100, 0);                                      // (2)
    } catch (ArithmeticException ae) {                           // (3)
      ae.printStackTrace();                                      // (4)
      System.out.println("Exception handled in main().");        // (5)
    }
    System.out.println("Exit main().");                          // (6)
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                        // (7)
      int average = computeAverage(totalSum, totalCount);        // (8)
      System.out.println("Average = " +                          // (9)
          totalSum + " / " + totalCount + " = " + average);
    } catch (IllegalArgumentException iae) {                     // (10)
      iae.printStackTrace();                                     // (11)
      System.out.println("Exception handled in printAverage().");  // (12)
    }
    System.out.println("Exit printAverage().");                  // (13)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                    // (14)
    return sum/count;                                            // (15)
  }
}
```

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:28)
        at Average3.printAverage(Average3.java:16)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

```
try {
  // ...
} catch (Exception e) {              // (1) Superclass shadows subclass
  System.out.println(e);
} catch (ArithmeticException e) {    // (2) Compile-time error: Unreachable code
  System.out.println(e);
}
```

```
try {
  throw new ArithmeticException();       // IOException never thrown in try block
} catch (IOException e) {                // Compile-time error: Unreachable code
  System.out.println(e);
}
```

```
int sum = 0;
try {
  sum = sumNumbers();                       // (1)
  // other actions
} finally {
  if (sum > 0) calculateAverage();          // (2)
}
```

```
public class Average4 {

  public static void main(String[] args) {
    printAverage(100, 20);                                         // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                          // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                             // (3)
      ae.printStackTrace();
      System.out.println("Exception handled in printAverage().");
    } finally {                                                    // (4)
      System.out.println("Finally done.");
    }
    System.out.println("Exit printAverage().");                    // (5)
  }
  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;
  }
}
```

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Exit printAverage().
Exit main().
```

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average4.computeAverage(Average4.java:24)
        at Average4.printAverage(Average4.java:10)
        at Average4.main(Average4.java:4)
Exception handled in printAverage().
Finally done.
Exit printAverage().
Exit main().
```

```
public class Average5 {

  public static void main(String[] args) {
    printAverage(100, 0);                                                 // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                                 // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } finally {                                                           // (3)
      System.out.println("Finally done.");
    }
    System.out.println("Exit printAverage().");
  }
  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;                                               // (4)
  }
}
```

```
Computing average.
Finally done.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average5.computeAverage(Average5.java:21)
        at Average5.printAverage(Average5.java:10)
        at Average5.main(Average5.java:4)
```

```
public class Average6 {

  public static void main(String[] args) {
    System.out.println("Value: " + printAverage(100, 20));         // (1)
    System.out.println("Exit main().");
  }

  public static int printAverage(int totalSum, int totalCount) {
    int average = 0;
    try {
      average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
      return average;                                             // (2)
    } finally {
      System.out.println("Finally done.");
      return average*2;                                           // (3)
    }
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;
  }
}
```

`throw` *object_reference_expression*`;`

```
throw new ArithmeticException("Integer division by 0");
```

```
public class Average7 {

  public static void main(String[] args) {
    printAverage(100, 0);            // Calling with 0 number of values.
  }

  public static void printAverage(int totalSum, int totalCount) {
    System.out.println("Entering printAverage().");
    try {
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                             // (1)
      ae.printStackTrace();
      System.out.println("Exception handled in printAverage().");
    } finally {                                                    // (2)
      System.out.println("Finally in printAverage().");
    }
    System.out.println("Exit printAverage().");                    // (3)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    if (count == 0)
      throw new ArithmeticException("Integer division by 0");      // (4)
    return sum/count;
  }
}
```

```
Entering printAverage().
Computing average.
java.lang.ArithmeticException: Integer division by 0
      at Average7.computeAverage(Average7.java:26)
      at Average7.printAverage(Average7.java:11)
      at Average7.main(Average7.java:5)
```

```
Exception handled in printAverage().
Finally in printAverage().
Exit printAverage().
```

... `throws` *ExceptionType*$_1$, *ExceptionType*$_2$, . . . , *ExceptionType*$_n$ `{ /* Body */ }`

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

```
// File: Average8.java
public class Average8 {
  public static void main(String[] args) {
    try {                                                          // (1)
      printAverage(100, 0);                                        // (2)
    } catch (IntegerDivisionByZero idbz) {                         // (3)
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                    // (4)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                            // (5)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {                               // (6)
    int average = computeAverage(totalSum, totalCount);            // (7)
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {                               // (8)
    System.out.println("Computing average.");
    if (count == 0)                                                // (9)
      throw new IntegerDivisionByZero();
    return sum/count;                                              // (10)
  }
}
```

```
Computing average.
IntegerDivisionByZero: Integer Division By Zero
        at Average8.computeAverage(Average8.java:27)
        at Average8.printAverage(Average8.java:17)
        at Average8.main(Average8.java:5)
Exception handled in main().
Finally done in main().
Exit main().
```

```
public static void main(String[] args) throws Exception {
  /* ... */
}

public static void printAverage(int totalSum, int totalCount) throws Exception {
  /* ... */
}
```

```
// New exception classes:
class FirstException    extends Exception { }
class SecondException   extends Exception { }
class ThirdException    extends Exception { }
class FourthException   extends Exception { }
class SubFirstException extends FirstException { }

// Superclass
class A {
  protected void compute()
      throws FirstException, SecondException, ThirdException { /* ... */ }  // (1)
}

// Subclass
class B extends A {
  @Override
  protected void compute()
      throws ThirdException, SubFirstException, NumberFormatException {     // (2)
    /* ... */
  }
}

//Subclass
class C extends A {
  @Override
  protected void compute()                             // Compile-time error at (3)
      throws FirstException, ThirdException, FourthException { /* ... */ }  // (3)
}
```

```
public class DemoClass {
  public static void main(String[] args) {
    int k=0;
    try {
      int i = 5/k;
    } catch (ArithmeticException e) {
      System.out.println("1");
    } catch (RuntimeException e) {
      System.out.println("2");
      return;
    } catch (Exception e) {
      System.out.println("3");
    } finally {
      System.out.println("4");
    }
    System.out.println("5");
  }
}
```

```
public class Exceptions {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    } finally {
      System.out.println("The end");
    }
  }
}
```

```
class A extends Throwable {}

class B extends A {}

public class RQ6A20 {
  public static void main(String[] args) throws A {
    try {
      action();
    } finally {
      System.out.println("Done.");
    } catch (A e) {
      throw e;
    }
  }

  public static void action() throws B {
    throw new B();
  }
}
```

```
public class Calculator {
  public static int average(int... values) {
    int result = 0;
    for (int i = 0; i < values.length; i++) {
      result += values[i];
    }
    return result/values.length;
  }
  public static void main(String[] args) {
    int value = 1;
    try {
      value = average();
    } catch (ArithmeticException e) {
      System.out.print("error");
    }
    System.out.print(value);
  }
}
```

```
public class PlayerException extends Exception {
  public PlayerException(String message) { super(message); }
}
```

```
public class Player {
  public static String reaction (String action)  throws PlayerException {
    if (action == null || action.isEmpty()) {
      throw new PlayerException("Invalid action");
    }
    return ">" + action;
  }
  public static void main(String[] args) {
    String message = null;
    try {
      message = reaction(message);
    } catch (PlayerException e) {
      message = e.getMessage();
    }
    System.out.print(message);
  }
}
```

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws Exception {
    if (name == null) throw new FileNotFoundException("invalid file name");
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

```
import java.io.*;
public class FileReader {
  public static void readFile(String name)  throws Exception {
    if (name != null) throw new FileNotFoundException("invalid file name");
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage()+" ");
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage()+" ");
    } finally {
      System.out.print("finally");
    }
    System.out.print(" the end");
  }
}
```

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws IOException {
    if (name == null) {
      throw new NullPointerException("invalid file name");
    } else {
      throw new IOException("file read not implemented");
    }
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
      return;
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

```
catch (RuntimeException rte) { // NOT RECOMMENDED!
  System.out.println(rte);
  System.out.println("Usage: java Average9 <sum of values> <no. of values>");
}
```

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

```
// File: Average9.java
public class Average9 {
  public static void main(String[] args) {
    try {                                                      // (1)
      int sum        = Integer.parseInt(args[0]);             // (2)
      int numOfValues = Integer.parseInt(args[1]);             // (3)
      printAverage(sum, numOfValues);                          // (4)
    } catch (ArrayIndexOutOfBoundsException aioob) {           // (5) uni-catch
      System.out.println(aioob);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>");
    } catch (NumberFormatException nfe) {                      // (6) uni-catch
      System.out.println(nfe);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>");
    } catch (IntegerDivisionByZero idbz) {                     // (7) uni-catch
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                // (8)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                        // (9)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    int average = computeAverage(totalSum, totalCount);
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    System.out.println("Computing average.");
    if (count == 0)
      throw new IntegerDivisionByZero();
    return sum/count;
  }
}
```

```
>java Average9 100 twenty
java.lang.NumberFormatException: For input string: "twenty"
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

```
>java Average9 100
java.lang.ArrayIndexOutOfBoundsException: 1
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

`catch` (*exception_type*$_1$|*exception_type*$_2$|…|*exception_type*$_k$ *parameter*) `{`
  statements
`}`

```
catch (ArrayIndexOutOfBoundsException |                  // (5) multi-catch
       NumberFormatException ep) {
  System.out.println(ep);
  System.out.println("Usage: java Average10 <sum of values> <no. of values>");
}
```

```
// File: Average10.java
public class Average10 {
  public static void main(String[] args) {
    try {                                                          // (1)
      int sum = Integer.parseInt(args[0]);                         // (2)
      int numOfValues = Integer.parseInt(args[1]);                 // (3)
      printAverage(sum, numOfValues);                       // (4)
    } catch (ArrayIndexOutOfBoundsException |               // (5) multi-catch
             NumberFormatException ep) {
      System.out.println(ep);
      System.out.println("Usage: java Average10 <sum of values> <no. of values>");
    } catch (IntegerDivisionByZero idbz) {                  // (6) uni-catch
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                             // (7)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                     // (8)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }
}
```

```
catch (IndexOutOfBoundsException |                   // Compile-time error!
       ArrayIndexOutOfBoundsException e) {
  // ...
}
```

```
try {
  // Assume appropriate code to throw the right exceptions.
} catch (NumberFormatException |
         IndexOutOfBoundsException e) {    // Parameter is final.
  e = new ArrayIndexOutOfBoundsException();// Compile-time error!
                                           // Cannot assign to final parameter e.
} catch (IntegerDivisionByZero idbz) {     // Parameter is effectively final.
  idbz.printStackTrace();
} catch (IOException ioe) {                // Parameter is not effectively final.
  ioe = new FileNotFoundException("No file.");
}
```

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionAnalysis {
  public static void main(String[] args) throws IOException {
    try {                                // (1)
      throw new FileNotFoundException(); // (2)
    } catch (IOException ex) {           // (3)
      try {                        // (4) Nested try statement
        throw ex;                  // (5) Can only rethrow FileNotFoundException
      } catch (EOFException se) {  // (6) Compile-time error: clause unreachable
        System.out.println("I am unreachable.");
      }
```

}

}

}

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class MorePreciseRethrow {
  public static void main(String[] args) {                        // (1)
    try {
      checkIt(1);
    } catch (FileNotFoundException fnfe) {
      System.out.println("Check that the file exits.");
    } catch (EOFException eofe) {
      System.out.println("Check the contents of the file.");
    } catch (IOException ioe) { // (2) mandatory with (3a), but compiler warning
                                //     that clause is unreachable with (3b).
      System.out.println("This should never occur.");
    }
  }
  public static void checkIt(int value) throws IOException {      // (3a)
//public static void checkIt(int value)                           // (3b)
//    throws FileNotFoundException, EOFException {
    try {                                                         // (4)
      switch (value) {
        case 1:
          throw new FileNotFoundException("File not found");
        case 2:
          throw new EOFException("End of file");
        default:
          System.out.println("OK");
      }
    } catch (IOException e) {                                      // (5)
      System.out.println(e.getMessage());
      e = new EOFException("End of file");          // (6) not effectively final,
                                                    //     requires (3a).
                                                    //     When commented out,
                                                    //     can use (3b).
      throw e;
    }
  }
}
```

File not found
Check the contents of the file.

File not found
Check that the file exits.

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class RethrowMe {
  public static void main(String[] args) throws EOFException {
    try {
      switch (1) {
        case 1: throw new FileNotFoundException("File not found");
        case 2: throw new EOFException("End of file");
        default: System.out.println("OK");
      }
    } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe);
    } catch (IOException ioe) {
      throw ioe;                                                // (1)
    }
  }

  public static void rethrowA() throws EOFException {
    try {
      // Empty try block.
    } catch (EOFException eofe) { // Compile-time error: exception not thrown
                                  //                     in try block.
      throw eofe;                                               // (2)
    }
  }

  public static void rethrowB() throws EOFException {
    try {
      throw new EOFException("End of file");
    } catch (EOFException eofe) {
      System.out.println(eofe);
    } catch (IOException ioe) {   // Compile-time warning: unreachable clause
      throw ioe;                                                // (3)
    }
  }
}
```

```
Throwable(Throwable cause)
Throwable(String msg, Throwable cause)
```

```
Throwable initCause(Throwable cause)
```

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionsInChain {
  public static void main(String[] args) {
    try {
      chainIt();
    } catch (Exception e) {                                        // (1)
      System.out.println("Exception chain: " + e);
      Throwable t = e.getCause();
      while (t != null) {
        System.out.println("Cause: " + t);
        t = t.getCause();
      }
    }
  }

  public static void chainIt() throws Exception {           // (2)
    try {
      throw new FileNotFoundException("File not found");
    } catch (FileNotFoundException e) {
      try {
        IOException ioe = new IOException("File error");
        ioe.initCause(e);                                   // (3)
        throw ioe;
      } catch (IOException ioe) {
        Exception ee = new Exception("I/O error", ioe);     // (4)
        throw ee;
      }
    }
  }
}
```

```
Exception chain: java.lang.Exception: I/O error
Cause: java.io.IOException: File error
Cause: java.io.FileNotFoundException: File not found
```

```
...
try {
  chainIt();
} catch (Exception e) {                                    // (1')
  e.printStackTrace();                                     // Print backtrace.
}
...
```

```
java.lang.Exception: I/O error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:23)
      at ExceptionsInChain.main(ExceptionsInChain.java:8)
Caused by: java.io.IOException: File error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:19)
      ... 1 more
Caused by: java.io.FileNotFoundException: File not found
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:16)
      ... 1 more
```

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class NaiveResourceUse {
  public static void main(String[] args)
      throws FileNotFoundException, EOFException, IOException {

    // Open the resource:
    var fis = new FileReader(args[0]);              // (1) FileNotFoundException
    var br = new BufferedReader(fis);

    // Use the resource:
    String textLine = br.readLine();                // (2) IOException
    if (textLine != null) {
      System.out.println(textLine);
    } else {
      throw new EOFException("Empty file.");       // (3) EOFException
    }

    // Close the resource:
    System.out.println("Closing the resource.");
    br.close();                                    // (4) IOException
  }
}
```

```
>java NaiveResourceUse EmptyFile.txt
Exception in thread "main" java.io.EOFException: Empty file.
        at NaiveResourceUse.main(NaiveResourceUse.java:20)
```

```
>java NaiveResourceUse Slogan.txt
Code Compile Compute
Closing the resource.
```

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TryWithoutARM {
  public static void main(String[] args) {
    BufferedReader br = null;
    try {
      // Open the resource:
      var fis = new FileReader(args[0]);             // (1) FileNotFoundException
      br = new BufferedReader(fis);

      // Use the resource:
      String textLine = br.readLine();               // (2) IOException
```

```
        if (textLine != null) {
          System.out.println(textLine);
        } else {
          throw new EOFException("Empty file.");       // (3) EOFException
        }
      } catch (FileNotFoundException | EOFException e) {
        e.printStackTrace();
      } catch (IOException ioe) {
        ioe.printStackTrace();
      } finally {                                      // (4)
        if (br != null) {
          try {
            System.out.println("Closing the resource.");
            br.close();                                // (5) IOException
          } catch(IOException ioe) {
            ioe.printStackTrace();
          }
        }
      }
    }
  }
```

```
>java TryWithoutARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithoutARM.main(TryWithoutARM.java:20)
Closing the resource.
```

```
>java TryWithoutARM Slogan.txt
Code Compile Compute
Closing the resource.
```

```
try ( resource_declaration_statement1; ... ; resource_declaration_statementm ) {
  statements
} catch (exception_type1 |...| exception_typek parameter1) { // multi-catch
  statements
}
...
  catch (exception_typen parametern) {                         // uni-catch
  statements
} finally {                                                     // finally
  statements
}
```

```
try (var fis = new FileReader(args[0]);        // (1) FileNotFoundException
     var br = new BufferedReader(fis)) {
  // ...
}
```

```
try (var br = new BufferedReader(new FileReader(args[0]))) {
  // ...
}
```

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.stream.Stream;

public class TryWithARM {
  public static void main(String[] args) {
    try (var fis = new FileReader(args[0]);         // (1) FileNotFoundExeception
         var br = new BufferedReader(fis)) {
      String textLine = br.readLine();              // (2) IOException
      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");      // (3) EOFException
      }
    } catch (FileNotFoundException | EOFException e) {
      e.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

```
>java TryWithARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithARM.main(TryWithARM.java:15)
```

```
>java TryWithARM Slogan.txt
Code Compile Compute
```

```
final ACResource1 acr1 = new ACResource1();      // (1)
ACResource2 acr2 = new ACResource2();            // (2) Effectively final at (3)
// ...
try (acr1; acr2) {                               // (3) Resource variables.
  // Use the resources.
} // Both resources closed on exit.
```

```
final ACResource1 acr1 = new ACResource1();                // (1)
// ...
try (acr1; ACResource2 acr2 = new ACResource2()) {         // (3)
  // Use the resources.
} catch (Exception ex) {                                   // (4)
  ex.printStackTrace();
} // Both resources closed on exit.
```

```
interface AutoCloseable {                      // java.lang package
  void close() throws Exception;
}

interface Closeable extends AutoCloseable {    // java.io package
  void close() throws IOException;             // Mandatory idempotent
}
```

```
// File: Gizmo.java
public class Gizmo implements AutoCloseable {
  private boolean closed = false;                // (1) Closed if true
  @Override
  public void close() {                          // (2) Idempotent
    System.out.println("Enter: close()");
    if (closed) {
      System.out.println("Already closed");
    } else {
      closed = true;
      System.out.println("Gizmo closed");
      System.out.println("Throwing IllegalArgumentException in close()");
      throw new IllegalArgumentException("thrown in close()"); // Suppressed
    }
    System.out.println("Exit: close()");     // Only executed if already closed.
  }
  public void compute() {                         // (3)
    System.out.println("Enter: compute()");
    System.out.println("Throwing ArithmeticException in compute()");
    throw new ArithmeticException("thrown in compute()");
  }
}
```

```
// File: GizmoTest.java
public class GizmoTest {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {              // (4)
      myGizmo.compute();
    } catch (Exception ex) {                       // (5)
      System.out.println("Printing stack trace in catch clause of main():");
      ex.printStackTrace();
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

```
// File: GizmoTest2.java
public class GizmoTest2 {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {
      myGizmo.compute();
    } catch (Exception ex) {                                        // (6)
      System.out.println("Exception caught in the catch clause of main():\n\t"
                         + ex);
      System.out.println("Printing suppressed exceptions "
                         + "in the catch clause of main():");
      Throwable[] supressedEx = ex.getSuppressed();                 // (7)
      for (Throwable t : supressedEx) {
        System.out.println("\t" + t);
      }
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Printing stack trace in catch clause of main():
java.lang.ArithmeticException: thrown in compute()
      at Gizmo.compute(Gizmo.java:21)
      at GizmoTest.main(GizmoTest.java:5)
      Suppressed: java.lang.IllegalArgumentException: thrown in close()
          at Gizmo.close(Gizmo.java:13)
          at GizmoTest.main(GizmoTest.java:6)
Finally: Done in main()
```

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Exception caught in the catch clause of main():
      java.lang.ArithmeticException: thrown in compute()
Printing suppressed exceptions in the catch clause of main():
      java.lang.IllegalArgumentException: thrown in close()
Finally: Done in main()
```

```
void addSuppressed(Throwable exception)
```

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ARMy {

  public static void methodA(String filename) throws FileNotFoundException {// (1)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodB(String filename)throws IOException {            // (2)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodC(String filename)throws FileNotFoundException { // (3)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
    catch (IOException ioe) { }
  }

  public static void methodD(String filename) throws FileNotFoundException {// (4)
    try (var fis = new FileReader(filename);
         var br = new BufferedReader(fis)) { }
  }

  public static void methodE(String filename) throws IOException {           // (5)
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodF(String filename) {                              // (6)
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodG(String filename)
                             throws IOException {                            // (7)
    try (FileReader fis = null; BufferedReader br = null) {
      fis = new FileReader(filename);
      br = new BufferedReader(fis);
    }
  }
}
```

```
import java.io.EOFException;
import java.io.IOException;

public class TryTwisting {
  public static void main(String[] args) {
    try {
      justDoIt();
    } catch (Exception t1) {
      System.out.println(t1);
      for (Throwable t : t1.getSuppressed())
        System.out.println("Suppressed: " + t);
    }
  }

  public static void justDoIt() throws Exception {
    IOException t2 = null;
    try {
      t2 = new IOException();
      throw t2;
    } finally {
      try {
        throw new EOFException();
      } catch (Exception t3) {
        t2.addSuppressed(t3);
      }
    }
  }
}
```

```
java.io.EOFException
Suppressed: java.io.IOException
```

```
java.io.EOFException
Suppressed: java.io.Exception
```

```
java.io.Exception
Suppressed: java.io.IOException
```

```
java.io.Exception
Suppressed: java.io.EOFException
```

```
java.io.IOException
Suppressed: java.io.EOFException
```

```
java.io.IOException
Suppressed: java.io.Exception
```

```
public class Truck implements AutoCloseable {
  public void close() {
    throw new Exception("Cannot close.");
  }

  public void load() {
    System.out.println("Loading truck.");
  }

  public static void main(String[] args) {
    try (var t1 = new Truck()) {
      t1.load();
    }
  }
}
```

```
void task1(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException | FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

```
void task2(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException e) {
    e = new IOException();
    throw e;
  } catch (FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

```
void task3() throws IOException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    e = new IOException();
    throw e;
  }
}
```

```
void task4() throws FileNotFoundException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    throw e;
  }
}
```

```
void task5() throws Exception {
  try {
    throw new FileNotFoundException();
  } catch (IOException e) {
    throw e;
  }
}
```

```
public class Resource implements AutoCloseable {
  public void action() { System.out.print("action "); }
  public void close() throws Exception { System.out.print("closure "); }
}
```

```
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (Exception ex) {
      System.out.print("error ");
    }
    System.out.print("the end ");
  }
}
```

```
import java.io.IOException;
public class Resource implements AutoCloseable {
  public void action() throws IOException {
    throw new IOException("action error ");
 }
  public void close() throws Exception {
    throw new Exception("closure error ");
 }
}
```

```
import java.io.IOException;
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (IOException ex) {
      System.out.print("IO ");
      System.out.print(ex.getMessage());
    } catch (Exception ex) {
      System.out.print("Other ");
      System.out.print(ex.getSuppressed()[0].getMessage());
    }
    System.out.print("the end ");
  }
}
```

```
public boolean equals(Object obj)
public final Class<?> getClass()
```

```
protected Object clone() throws CloneNotSupportedException
```

```
@Deprecated(since="9") protected void finalize() throws Throwable
```

```
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

```
// File: ObjectMethods.java
class MyClass implements Cloneable {
  @Override
  public MyClass clone() {
    MyClass obj = null;
    try { obj = (MyClass) super.clone(); }  // Calls overridden method.
    catch (CloneNotSupportedException e) { System.out.println(e);}
    return obj;
  }
}
//______________________________________________________________________________
public class ObjectMethods {
  public static void main(String[] args) {
    // Two objects of MyClass.
    MyClass obj1 = new MyClass();
    MyClass obj2 = new MyClass();

    // Two strings.
    String str1 = new String("WhoAmI");
    String str2 = new String("WhoAmI");

    System.out.println("str1: " + str1.toString());
    System.out.println("str2: " + str2.toString() + "\n");

    // Method hashCode() overridden in String class.
    // Strings that are equal have the same hash code
    // Method hashCode() overridden in String class.
    // Strings that are equal have the same hash code.
    System.out.println("hash code for str1: " + str1.hashCode());
    System.out.println("hash code for str2: " + str2.hashCode() + "\n");

    // Hash codes are different for different MyClass objects.
    System.out.println("hash code for MyClass obj1: " + obj1.hashCode());
    System.out.println("hash code for MyClass obj2: " + obj2.hashCode()+"\n");

    // Method equals() overridden in the String class.
    System.out.println("str1.equals(str2): " + str1.equals(str2));
    System.out.println("str1 == str2:      " + (str1 == str2) + "\n");

    // Method equals() from the Object class called.
    System.out.println("obj1.equals(obj2): " + obj1.equals(obj2));
    System.out.println("obj1 == obj2:      " + (obj1 == obj2) + "\n");

    // The runtime object that represents the class of an object.
    Class<? extends String> rtStringClass  = str1.getClass();
    Class<? extends MyClass> rtMyClassClass = obj1.getClass();
    // The name of the class represented by the runtime object.
    System.out.println("Class for str1: " + rtStringClass);
    System.out.println("Class for obj1: " + rtMyClassClass + "\n");

    // The toString() method is overridden in the String class.
    String textRepStr = str1.toString();
    String textRepObj = obj1.toString();
    System.out.println("Text representation of str1: " + textRepStr);
    System.out.println("Text representation of obj1: " + textRepObj + "\n");

    // Shallow copying of arrays.
    MyClass[] array1 = {new MyClass(), new MyClass(), new MyClass()};
    MyClass[] array2 = array1.clone();
    // Array objects are different, but share the element objects.
    System.out.println("array1 == array2:        " + (array1 == array2));
    for(int i = 0; i < array1.length; i++) {
      System.out.println("array1[" + i + "] == array2[" + i + "] : " +
                         (array1[i] == array2[i]));
    }
    System.out.println();

    // Clone an object of MyClass.
    MyClass obj3 = obj1.clone();
    System.out.println("hash code for cloned MyClass obj3: " + obj3.hashCode());
    System.out.println("obj1 == obj3: " + (obj1 == obj3));
  }
}
```

```
str1: WhoAmI
str2: WhoAmI

hash code for str1: -1704812257
hash code for str2: -1704812257
hash code for MyClass obj1: 2036368507
hash code for MyClass obj2: 1785210046

str1.equals(str2): true
str1 == str2:      false

obj1.equals(obj2): false
obj1 == obj2:      false

Class for str1: class java.lang.String
Class for obj1: class MyClass

Text representation of str1: WhoAmI
Text representation of obj1: MyClass@7960847b

array1 == array2:        false
array1[0] == array2[0] : true
array1[1] == array2[1] : true
array1[2] == array2[2] : true

hash code for cloned MyClass obj3: 1552787810
obj1 == obj3: false
```

```
Character charObj1   = '\n';
Boolean   boolObj1   = true;
Integer   intObj1    = 2020;
Double    doubleObj1 = 3.14;
```

```
static WrapperType valueOf(type v)

Character charObj1   = Character.valueOf('\n');
Boolean   boolObj1   = Boolean.valueOf(true);
Integer   intObj1    = Integer.valueOf(2020);
Double    doubleObj1 = Double.valueOf(3.14);
```

```
static WrapperType valueOf(String str)
```

```
Boolean boolObj4   = Boolean.valueOf("false");
Integer intObj3    = Integer.valueOf("1949");
Double  doubleObj2 = Double.valueOf("3.14");
Double  doubleObj3 = Double.valueOf("Infinity");
```

```
static IntegerWrapperType valueOf(String str, int base)
                                  throws NumberFormatException
```

```
Byte    byteObj1  = Byte.valueOf("1010", 2);     // Decimal value 10
Short   shortObj2 = Short.valueOf("12", 8);      // Decimal value 10
Short   shortObj3 = Short.valueOf("012", 8);     // Decimal value 10
Short   shortObj4 = Short.valueOf("\012", 8);    // NumberFormatException
Integer intObj4   = Integer.valueOf("-a", 16);   // Decimal value -10
Integer intObj6   = Integer.valueOf("-0xa", 16); // NumberFormatException
Long    longObj2  = Long.valueOf("-a", 16);      // Decimal value -10L
```

```
String charStr   = charObj1.toString();        // "\n"
String boolStr   = boolObj2.toString();        // "true"
String intStr    = intObj1.toString();         // "2020"
String doubleStr = doubleObj1.toString();      // "3.14"
```

`static String toString(`*`type`*` v)`

```
String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2020);    // "2020"
String doubleStr2 = Double.toString(3.14);     // "3.14"
```

```
char    c = charObj1;                                   // '\n'
boolean b = boolObj2;                                   // true
int     i = intObj1;                                    // 2020
double  d = doubleObj1;                                 // 3.14
```

```
char    c = charObj1.charValue();             // '\n'
boolean b = boolObj2.booleanValue();          // true
int     i = intObj1.intValue();               // 2020
double  d = doubleObj1.doubleValue();         // 3.14
```

int compareTo(*Type* obj2)

int compareTo(*Type* obj2)

```
// Comparisons based on objects created earlier
Character charObj2   = 'a';
int result1 = charObj1.compareTo(charObj2);      // result1 < 0
int result2 = intObj1.compareTo(intObj3);        // result2 > 0
int result3 = doubleObj1.compareTo(doubleObj2);  // result3 == 0
int result4 = doubleObj1.compareTo(intObj1);     // Compile-time error!
```

`boolean equals(Object obj2)`

```
// Comparisons based on objects created earlier
boolean charTest   = charObj1.equals(charObj2);      // false
boolean boolTest   = boolObj2.equals(Boolean.FALSE); // false
boolean intTest    = intObj1.equals(intObj3);        // true
boolean doubleTest = doubleObj1.equals(doubleObj2);  // true
boolean test       = intObj1.equals(Long.valueOf(2020L)); // false. Not same type.
```

```
// Reference and object equality
Byte bRef1 = 10;
Byte bRef2 = 10;
System.out.println(bRef1 == bRef2);            // true
System.out.println(bRef1.equals(bRef2));       // true

Integer iRef1 = 1000;
Integer iRef2 = 1000;
System.out.println(iRef1 == iRef2);            // false, values not in [-128, 127]
System.out.println(iRef1.equals(iRef2));       // true
```

```
int hashCode()

int index = charObj1.hashCode();                    // 10 ('\n')
```

NumericWrapperType`.MIN_VALUE`
NumericWrapperType`.MAX_VALUE`

```
byte   minByte   = Byte.MIN_VALUE;     // -128
int    maxInt    = Integer.MAX_VALUE;  // 2147483647
double maxDouble = Double.MAX_VALUE;   // 1.7976931348623157e+308
```

`byte`	`byteValue()`
`short`	`shortValue()`
`int`	`intValue()`
`long`	`longValue()`
`float`	`floatValue()`
`double`	`doubleValue()`

```
Byte    byteObj2   = (byte)16;                  // Cast mandatory
Integer intObj5    = 42030;
Double  doubleObj4 = Math.PI;

short  shortVal  = intObj5.shortValue();        // (1)
long   longVal   = byteObj2.longValue();
int    intVal    = doubleObj4.intValue();       // (2) Truncation
double doubleVal = intObj5.doubleValue();
```

```
static type parseType(String str) throws NumberFormatException
```

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2020");       // parseInt, not parseInteger
int    value3 = Integer.parseInt("7UP");        // NumberFormatException.
double value4 = Double.parseDouble("3.14");
double value5 = Double.parseDouble("Infinity");
```

```
type parseType(String str, int base) throws NumberFormatException
```

```
byte  value6  = Byte.parseByte("1010", 2);        // Decimal value 10
short value7  = Short.parseShort("12", 8);        // Decimal value 10
short value8  = Short.parseShort("012", 8);       // Decimal value 10
short value9  = Short.parseShort("\012", 8);      // NumberFormatException
int   value10 = Integer.parseInt("-a", 16);       // Decimal value -10
int   value11 = Integer.parseInt("-0xa", 16);     // NumberFormatException
long  value12 = Long.parseLong("-a", 16);         // Decimal value -10L
```

```
static String toBinaryString(int i)
static String toHexString(int i)
static String toOctalString(int i)
static String toString(int i, int base)
static String toString(int i)
```

```
public class IntegerRepresentation {
  public static void main(String[] args) {
    int positiveInt = +41;    // 0b101001, 051, 0x29
    int negativeInt = -41;    // 0b11111111111111111111111111010111, -0b101001,
                              // 037777777727, -051, 0xffffffd7, -0x29
    System.out.println("Text representation for decimal value: " + positiveInt);
    integerStringRepresentation(positiveInt);
    System.out.println("Text representation for decimal value: " + negativeInt);
    integerStringRepresentation(negativeInt);
  }

  public static void integerStringRepresentation(int i) {
    System.out.println("    Binary:    " + Integer.toBinaryString(i));
    System.out.println("    Octal:     " + Integer.toOctalString(i));
    System.out.println("    Hex:       " + Integer.toHexString(i));
    System.out.println("    Decimal:   " + Integer.toString(i));

    System.out.println("    Using toString(int i, int base) method:");
    System.out.println("    Base 2:    " + Integer.toString(i, 2));
    System.out.println("    Base 8:    " + Integer.toString(i, 8));
    System.out.println("    Base 16:   " + Integer.toString(i, 16));
    System.out.println("    Base 10:   " + Integer.toString(i, 10));
  }
}
```

```
Text representation for decimal value: 41
    Binary:    101001
    Octal:     51
    Hex:       29
    Decimal:   41
    Using toString(int i, int base) method:
    Base 2:    101001
    Base 8:    51
    Base 16:   29
    Base 10:   41
Text representation for decimal value: -41
    Binary:    11111111111111111111111111010111
    Octal:     37777777727
    Hex:       ffffffd7
    Decimal:   -41
    Using toString(int i, int base) method:
    Base 2:    -101001
    Base 8:    -51
    Base 16:   -29
    Base 10:   -41
```

```
Character.MIN_VALUE
Character.MAX_VALUE
```

```
static int     getNumericValue(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
static boolean isTitleCase(char ch)
static boolean isDigit(char ch)
static boolean isLetter(char ch)
static boolean isLetterOrDigit(char ch)
static char    toUpperCase(char ch)
static char    toLowerCase(char ch)
static char    toTitleCase(char ch)
```

```
char ch = 'a';
if (Character.isLowerCase(ch)) ch = Character.toUpperCase(ch); // A
```

```
static boolean parseBoolean(String str)
```

```
boolean b1 = Boolean.parseBoolean("TRUE");      // true.
boolean b2 = Boolean.parseBoolean("true");      // true.
boolean b3 = Boolean.parseBoolean("false");     // false.
boolean b4 = Boolean.parseBoolean("FALSE");     // false.
boolean b5 = Boolean.parseBoolean("not true");  // false.
boolean b6 = Boolean.parseBoolean("null");      // false.
boolean b7 = Boolean.parseBoolean(null);        // false.
```

```
public class RQ200A70 {
  public static void main(String[] args) {
    Integer i = Integer.valueOf(-10);
    Integer j = Integer.valueOf(-10);
    Integer k = -10;
    System.out.print((i==j) + "|");
    System.out.print(i.equals(j) + "|");
    System.out.print((i==k) + "|");
    System.out.print(i.equals(k));
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Integer i1 = 10;
    Integer i2 = 10;
    int i3 = 10;
    Integer x1 = i1*i2*i3;
    Integer x2 = i1*i2*i3;
    int x3 = i1*i2*i3;
    String result = (i1 == i2) ? "A" : "";
    result += (i1 == i3) ? "B" : "";
    result += (x1 == x2) ? "C" : "";
    result += (x1 == x3) ? "D" : "";
    System.out.print(result);
  }
}
```

```
String str1 = "You cannot change me!";
```

```
int strLength = "You cannot change me!".length(); // 21
```

```
String str2 = "You cannot change me!";      // Interned string assigned.
String str3 = "You cannot" + " change me!"; // Compile-time constant expression
String can1 = 7 + "Up";         // Value of compile-time constant expression: "7Up"
String can2 = "7Up";            // "7Up"
boolean r = can1 == can2;       // true
```

```
String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

```
String(char[] value)
String(char[] value, int offset, int count)
String(StringBuilder builder)
```

```
String str4 = new String("You cannot change me!");
```

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
StringBuilder strBuilder = new StringBuilder("abba");
//...
String byteStr  = new String(bytes);      // Using array of bytes: "abba"
String charStr  = new String(characters); // Using array of chars: "abba"
String buildStr = new String(strBuilder); // Using string builder: "abba"
```

`strX.intern() == strY.intern()` *is* `true` *if and only if* `strX.equals(strY)` *is* `true`.

```
// File: StringConstruction.java
class Auxiliary {
  static String str1 = "You cannot change me!";          // Interned
}
//______________________________________________________________________________
public class StringConstruction {

  static String str1 = "You cannot change me!";          // Interned

  public static void main(String[] args) {
    String emptyStr = new String();                      // ""
    System.out.println("emptyStr: \"" + emptyStr + "\"");

    String str2 = "You cannot change me!";               // Interned
    String str3 = "You cannot" + " change me!";          // Interned
    String str4 = new String("You cannot change me!");   // New String object

    String words = " change me!";
    String str5 = "You cannot" + words;                  // New String object

    System.out.println("str1 == str2:      " +  (str1 == str2));     // (1) true
    System.out.println("str1.equals(str2): " +  str1.equals(str2));  // (2) true

    System.out.println("str1 == str3:      " + (str1 == str3));      // (3) true
    System.out.println("str1.equals(str3): " + str1.equals(str3));   // (4) true

    System.out.println("str1 == str4:      " + (str1 == str4));      // (5) false
    System.out.println("str1.equals(str4): " + str1.equals(str4));   // (6) true

    System.out.println("str1 == str5:      " + (str1 == str5));      // (7) false
    System.out.println("str1.equals(str5): " + str1.equals(str5));   // (8) true

    System.out.println("str1 == Auxiliary.str1:      " +
                       (str1 == Auxiliary.str1));        // (9) true
    System.out.println("str1.equals(Auxiliary.str1): " +
                        str1.equals(Auxiliary.str1));    // (10) true

    System.out.println("\"You cannot change me!\".length(): " +
                       "You cannot change me!".length());// (11) 21

    System.out.println("str1.intern() == str4.intern(): " +
                       (str1.intern() == str4.intern()));// (12) true
    System.out.println("str4 == str4.intern(): " +
                       (str4 == str4.intern()));         // (13) false
  }
}
```

```
emptyStr: ""
str1 == str2:      true
str1.equals(str2): true
str1 == str3:      true
```

```
str1.equals(str3): true
str1 == str4:      false
str1.equals(str4): true
str1 == str5:      false
str1.equals(str5): true
str1 == Auxiliary.str1:      true
str1.equals(Auxiliary.str1): true
"You cannot change me!".length(): 21
str1.intern() == str4.intern(): true
str4 == str4.intern(): false
```

```
CharSequence subSequence(int start, int end)
```

```
static int compare(CharSequence cs1, CharSequence cs2)
```

`char charAt(int index)`	From the `CharSequence` interface (p. 444).
`char[] toCharArray()`	

```
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

`IntStream chars()`	From the `CharSequence` interface (p. 444).
`int length()`	From the `CharSequence` interface (p. 444).
`boolean isEmpty()`	From the `CharSequence` interface (p. 444).

```
public class ReadingCharsFromString {
  public static void main(String[] args) {
    int[] frequencyData = new int [Character.MAX_VALUE];    // (1)
    String str = "You cannot change me!";                   // (2)

    // Count the frequency of each character in the string.
    for (int i = 0; i < str.length(); i++) {                // (3)
```

```
      try {
        frequencyData[str.charAt(i)]++;                     // (4)
      } catch(StringIndexOutOfBoundsException e) {
        System.out.println("Index error detected: "+ i +" not in range.");
      }
    }

    // Print the character frequency.
    System.out.println("Character frequency for string: \"" + str + "\"");
    for (int i = 0; i < frequencyData.length; i++) {
      if (frequencyData[i] != 0)
        System.out.println((char)i + " (code "+ i +"): " + frequencyData[i]);
    }

    System.out.println("Copying into a char array:");
    char[] destination = new char [str.length() - 3];       // 3 characters less.
    str.getChars( 0,            7, destination, 0);         // (5) "You can"
    str.getChars(10, str.length(), destination, 7);         // (6) " change me!"
                                                            // "not" not copied.
    // Print the character array.
    for (int i = 0; i < destination.length; i++) {
      System.out.print(destination[i]);
    }
    System.out.println();
  }
}
```

```
Character Frequency for string: "You cannot change me!"
  (code 32): 3
! (code 33): 1
Y (code 89): 1
a (code 97): 2
c (code 99): 2
e (code 101): 2
g (code 103): 1
h (code 104): 1
m (code 109): 1
n (code 110): 3
o (code 111): 2
t (code 116): 1
u (code 117): 1
Copying into a char array:
You can change me!
```

```
    frequencyData[str.charAt(i)]++;                    // (4)
```

```
boolean test = 'a' < 'b';    // true since 0x61 < 0x62
boolean equals(Object obj)
boolean equalsIgnoreCase(String str2)
```

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

boolean b1 = strA.equals(strB);             // false
boolean b2 = strA.equalsIgnoreCase(strB);   // true

String str1 = "abba";
String str2 = "aha";

int compVal1 = str1.compareTo(str2);         // negative value => str1 < str2
```

```
String toUpperCase()
String toUpperCase(Locale locale)

String toLowerCase()
String toLowerCase(Locale locale)
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

String strC = strA.toLowerCase();  // Case conversion => New String object:
                                   // "the case was thrown out of court"
String strD = strB.toLowerCase();  // No case conversion => Same String object
String strE = strA.toUpperCase();  // Case conversion => New String object:
                                   // "THE CASE WAS THROWN OUT OF COURT"

boolean test1 = strC == strA;      // false
boolean test2 = strD == strB;      // true
boolean test3 = strE == strA;      // false
```

```
String billboard = "Just";
billboard.concat(" lost in space."); // (1) Returned reference value not stored.
System.out.println(billboard);       // (2) "Just"
billboard = billboard.concat(" advertise").concat(" here.");  // (3) Chaining.
System.out.println(billboard);       // (4) "Just advertise here."
```

```
String motto = new String("Program once");       // (1)
motto += ", execute everywhere.";                 // (2)
motto  = motto.concat(" Don't bet on it!");     // (3)
```

"Program once, execute everywhere. Don't bet on it!"

```
String banner = "Let's ace that Exam! ";
String bigBanner = banner.repeat(3);
System.out.println("|" + bigBanner + "|");
```

```
static String join(CharSequence delimiter, CharSequence... elements)
static String join(CharSequence delimiter,
                   Iterable<? extends CharSequence> elements)
```

```
// (1) Joining individual String objects:
String dateStr = String.join("/", "2014", "January", "11");
System.out.println(dateStr);                    // 2014/January/11
```

```
// (2) Joining elements in a StringBuilder array:
StringBuilder left          = new StringBuilder("Left");
StringBuilder right         = new StringBuilder("Right");
StringBuilder[] strBuilders = { left, right, left };
```

```
String march = String.join("-->", strBuilders);
System.out.println(march);                    // Left-->Right-->Left
```

```
// (3) Joining elements in a StringBuilder list:
ArrayList<StringBuilder> sbList = new ArrayList<>();
sbList.add(right); sbList.add(null); sbList.add(left); sbList.add(null);
String resultStr = "[" + String.join(", ", sbList) + "]";
System.out.println(resultStr);                // [Right, null, Left, null]
```

```
// (4) Joining elements in a CharSequence list:
ArrayList<CharSequence> charSeqList = new ArrayList<>();
charSeqList.add(right); charSeqList.add(left);     // Add StringBuilder objects.
charSeqList.add("Right"); charSeqList.add("Left"); // Add String objects.
String resultStr2 = "<" + String.join("; ", charSeqList) + ">";
System.out.println(resultStr2);               // <Right; Left; Right; Left>
```

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
```

```
boolean contains(CharSequence cs)
boolean startsWith(String prefix)
boolean startsWith(String prefix, int index)
boolean endsWith(String suffix)
```

```
String funStr = "Java Jives";
//               0123456789

int jInd1a = funStr.indexOf('J');                // 0
int jInd1b = funStr.indexOf('J', 1);             // 5
int jInd2a = funStr.lastIndexOf('J');          // 5
int jInd2b = funStr.lastIndexOf('J', 4);       // 0

String banner = "One man, One vote";
//               01234567890123456

int subInd1a = banner.indexOf("One");          // 0
int subInd1b = banner.indexOf("One", 3);       // 9
int subInd2a = banner.lastIndexOf("One");      // 9
int subInd2b = banner.lastIndexOf("One", 10); // 9
int subInd2c = banner.lastIndexOf("One", 8);  // 0
int subInd2d = banner.lastIndexOf("One", 2);  // 0

String newStr    = funStr.replace('J', 'W');      // "Wava Wives"
String newBanner = banner.replace("One", "No");   // "No man, No vote"
boolean found1   = banner.contains("One");        // true
boolean found2   = newBanner.contains("One");     // false

String song = "Start me up!";
//             012345677890
boolean found3    = song.startsWith("Start");     // true
boolean notFound1 = song.startsWith("start");     // false
boolean found4    = song.startsWith("me", 6);     // true
boolean found5    = song.endsWith("up!");         // true
boolean notFound2 = song.endsWith("up");          // false
```

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

`CharSequence subSequence(int start, int end)` From the `CharSequence` interface (p. 444)

```
System.out.println(Character.isWhitespace('\t'));  // true
System.out.println(Character.isWhitespace('\n'));  // true
System.out.println(Character.isWhitespace('a'));   // false
System.out.println("".isBlank());          // true
System.out.println(" \t  ".isBlank());     // true
```

```
String utopia = "\t\n  Java Nation \n\t  ";
System.out.println(utopia.strip().equals("Java Nation"));               // true
System.out.println(utopia.stripLeading().equals("Java Nation \n\t  ")); // true
System.out.println(utopia.stripTrailing().equals("\t\n  Java Nation")); // true
```

```
String utopia2 = "\t\n  Java Nation \n\t  ";
utopia2 = utopia2.trim();                       // "Java Nation"
utopia2 = utopia2.substring(5);                 // "Nation"
String radioactive = utopia2.substring(3,6);    // "ion"
```

`String toString()`	From the `CharSequence` interface (p. 444).

```
static String valueOf(Object obj)
static String valueOf(char[] charArray)
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

```
String anonStr   = String.valueOf("Make me a string.");      // "Make me a string."
String charStr   = String.valueOf(new char[] {'a', 'h', 'a'});// "aha"
String boolTrue  = String.valueOf(true);                       // "true"
String doubleStr = String.valueOf(Math.PI);                  // "3.141592653589793"
```

```
<R> R transform(Function<? super String,? extends R> f)
```

```
String message = "Take me to your leader!";
String tongueSpeake = message.transform(s ->
    new StringBuilder(s).reverse().toString().toUpperCase());
System.out.println(tongueSpeake); // !REDAEL RUOY OT EM EKAT
```

```
String cmds = " Attention!\r  Quick march!\n\n   Eyes left!"; // (0)
String str1 = cmds.indent(0);                                // (1)
String str2 = cmds.indent(3);                                // (2)
String str3 = cmds.indent(-1);                               // (3)
String str4 = cmds.indent(-2);                               // (4)
String str5 = cmds.indent(-3);                               // (5)
```

```
(0) cmds ==> " Attention!\r   Quick march!\n\n   Eyes left!"
(1) str1 ==> " Attention!\n   Quick march!\n\n   Eyes left!\n"
(2) str2 ==> "    Attention!\n     Quick march!\n    \n      Eyes left!\n"
(3) str3 ==> "Attention!\n Quick march!\n\n  Eyes left!\n"
(4) str4 ==> "Attention!\nQuick march!\n\n Eyes left!\n"
(5) str5 ==> "Attention!\nQuick march!\n\nEyes left!\n"
```

```
static String format(String format, Object... args)
```

```
String formatted(Object... args)
String formattedStr = String.format("Formatted values|%5d|%8.3f|%5s|",
                                    2020, Math.PI, "Hi");
System.out.println(formattedStr);  // Formatted values| 2020|  3.142|   Hi|
formattedStr = formattedStr.toUpperCase();
System.out.println(formattedStr);  // FORMATTED VALUES| 2020|  3.142|   HI|
String formattedStr1 = "Formatted values|%5d|%8.3f|%5s|"
                          .formatted(2020, Math.PI, "Hi");
System.out.println(formattedStr1); // Formatted values| 2020|   3.142|   Hi|
String sql1 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';
""";
```

```
sql1 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n";
```

```
SELECT *
FROM Programmers
WHERE Language = 'Java';

_
```

```
String badBlock = """       // (1) Compile-time error!
Not a good start.
""";
```

```
String emptyBlock = """
""";
```

```
String sql2 = """
SELECT *
FROM Programmers\nWHERE Language = 'Java';
""";
```

```
sql2 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n"
```

```
String sql3 = """
SELECT *
FROM Programmers\n
WHERE Language = 'Java';
""";
```

```
sql3 ==> "SELECT *\nFROM Programmers\n\nWHERE Language = 'Java';\n"
```

```
SELECT *
FROM Programmers

WHERE Language = 'Java';

_
```

```
String sql4 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';"""; // No line terminator. Closing delimiter ends block.
```

```
sql4 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';"
```

```
System.out.print(sql4);
```

```
SELECT *
FROM Programmers
WHERE Language = 'Java';
```

```
String sql5 = """
SELECT * \
FROM Programmers \
WHERE Language = 'Java';
""";
```

```
sql5 ==> "SELECT * FROM Programmers WHERE Language = 'Java';\n"
```

```
SELECT * FROM Programmers WHERE Language = 'Java';
_
```

```
FROM Programmers   \s     """;                   // (1) No line termination.
// sql6 ==> "SELECT *\nFROM Programmers    "

String sql7 = "WHERE Language = 'Java';\n";
System.out.print(sql6 + sql7);
```

```
SELECT *
FROM Programmers    WHERE Language = 'Java';
_
```

```
String fact = """
The sequence \"""
has "special" meaning in a text block.
""";
// fact ==> "The sequence \"\"\"\nhas \"special\" meaning in a text block.\n"
System.out.print(fact);
```

```
The sequence """
has "special" meaning in a text block.

_
```

```
String query = """
SELECT *
FROM %s
WHERE %s = '%s';
""".formatted("Customers", "Country", "Norway");
System.out.print(query);
```

```
SELECT *
FROM Customers
WHERE Country = 'Norway';

_
```

```
tb1 ==> "    KEEP\n      IT\n         SIMPLE\n"
```

```
tb2 ==> "  KEEP\n    IT\n      SIMPLE\n"
```

```
tb3 ==> "  KEEP\nIT\n  SIMPLE\n"
```

```
tb4 ==> "  KEEP\n    IT\nSIMPLE"
```

```
tb5 ==> "KEEP\nIT\n  SIMPLE"
```

```
StringBuilder(String str)
StringBuilder(CharSequence charSeq)
StringBuilder(int initialCapacity)
```

```
StringBuilder strBuilder1 = new StringBuilder("Phew!");  // "Phew!", capacity 21
StringBuilder strBuilder2 = new StringBuilder(10);       // "", capacity 10
StringBuilder strBuilder3 = new StringBuilder();         // "", capacity 16
```

`char charAt(int index)` From the `CharSequence` interface (p. 444).

`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`

`IntStream chars()`	From the `CharSequence` interface (p. 444).
`int length()`	From the `CharSequence` interface (p. 444).

```
StringBuilder strBuilder = new StringBuilder("Java");      // "Java", capacity 20
char charFirst = strBuilder.charAt(0);                     // 'J'
char charLast  = strBuilder.charAt(strBuilder.length()-1); // 'a'
```

```
int indexOf(String substr)
int indexOf(String substr, int fromIndex)

int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

```
StringBuilder banner2 = new StringBuilder("One man, One vote");
//                                          01234567890123456

int subInd1a = banner2.indexOf("One");         // 0
int subInd1b = banner2.indexOf("One", 3);      // 9
int subInd2a = banner2.lastIndexOf("One");     // 9
int subInd2b = banner2.lastIndexOf("One", 10); // 9
int subInd2c = banner2.lastIndexOf("One", 8);  // 0
int subInd2d = banner2.lastIndexOf("One", 2);  // 0
```

`String substring(int startIndex)`
`String substring(int startIndex, int endIndex)`

`CharSequence subSequence(int start, int end)` From the `CharSequence` interface (p. 444).

```
StringBuilder bookName = new StringBuilder("Java Gems by Anonymous");
//                                          01234567890123456789012
String title  = bookName.substring(0,9);    // "Java Gems"
String author = bookName.substring(13);     // "Anonymous"
```

`String toString()`	From the `CharSequence` interface (p. 444).

```
StringBuilder strBuilder = new StringBuilder("Build or not to build.");
String fromBuilder = strBuilder.toString();     // "Build or not to build."
```

```
int compareTo(StringBuilder anotherSB)
```

```
StringBuilder sb1 = new StringBuilder("abba");
StringBuilder sb2 = new StringBuilder("aha");

int compValue1 = sb1.compareTo(sb2);         // negative value => sb1 < sb2
int compValue2 = sb2.compareTo(sb1);         // positive value => sb2 > sb1
boolean isEqual = sb1.compareTo(sb2) == 0;               // false
boolean flag    = sb1.toString().equals(sb2.toString()); // false
```

```
StringBuilder append(Object obj)
StringBuilder append(String str)
StringBuilder append(CharSequence charSeq)
StringBuilder append(CharSequence charSeq, int start, int end)
StringBuilder append(char[] charArray)
StringBuilder append(char[] charArray, int offset, int length)
StringBuilder append(char c)
StringBuilder append(boolean b)
StringBuilder append(int i)
StringBuilder append(long l)
StringBuilder append(float f)
StringBuilder append(double d)
StringBuilder insert(int offset, Object obj)
StringBuilder insert(int dstOffset, CharSequence seq)
StringBuilder insert(int dstOffset, CharSequence seq, int start, int end)
StringBuilder insert(int offset, String str)
StringBuilder insert(int offset, char[] charArray)
StringBuilder insert(int offset, type c)
```

```
void setCharAt(int index, char ch)
```

```
StringBuilder replace(int start, int end, String replacement)
StringBuilder deleteCharAt(int index)
StringBuilder delete(int start, int end)
StringBuilder builder = new StringBuilder("banana split");   // "banana split"
builder.delete(4,12);                                         // "bana"
builder.append(42);                                           // "bana42"
builder.insert(4,"na");                                       // "banana42"
builder.reverse();                                            // "24ananab"
builder.deleteCharAt(builder.length()-1);                     // "24anana"
builder.replace(0, 2, "b");                                   // "banana"
builder.append('s');                                          // "bananas"
builder.delete(4,12).append(42).insert(4,"na").reverse()
       .deleteCharAt(builder.length()-1).replace(0, 2, "b")
       .append('s');                                        // "bananas"
```

```
((((((builder.delete(4,12)).append(42)).insert(4,"na")).reverse())
      .deleteCharAt(builder.length()-1)).replace(0, 2, "b"))
      .append('s');                                          // "bananas"
```

```
String theChosen = "U";
String str1 = 4 + theChosen + "Only";           // (1) Non-constant expression.
String str2 = new StringBuilder().
                  append(4).append(theChosen).append("Only").toString(); // (2)
```

```
void ensureCapacity(int minCapacity)
void setLength(int newLength)
```

```
public class StringBuilderCapacity {
  public static void main(String[] args) {
    StringBuilder builder = new StringBuilder("No strings attached!");
    System.out.println("Builder contents: " + builder);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());
    System.out.println("Ensure capacity of 40");
    builder.ensureCapacity(40);
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Trim to size");
    builder.trimToSize();
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Set length to 10");
    builder.setLength(10);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder contents: " + builder);
    System.out.println("Set length to 0");
    builder.setLength(0);
    System.out.println("Builder is empty: " + (builder.length() == 0));
  }
}
```

```
Builder contents: No strings attached!
Builder length:   20
Builder capacity: 36
Ensure capacity of 40
Builder capacity: 74
Trim to size
Builder length:   20
Builder capacity: 20
Set length to 10
Builder length:   10
Builder contents: No strings
Set length to 0
Builder is empty: true
```

```
class MyClass {
  public static void main(String[] args) {
    String str1 = "str1";
    String str2 = "str2";
    String str3 = "str3";

    str1.concat(str2);
    System.out.println(str3.concat(str1));
  }
}
```

```
public class RefEq {
  public static void main(String[] args) {
    String s = "ab" + "12";
    String t = "ab" + 12;
    String u = new String("ab12");
    System.out.println((s==t) + " " + (s==u));
  }
}
```

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

```
public class FunCharSeq {
  private static void putO(String s1) {
    s1 = s1.trim();
    s1 += "O";
  }

  public static void main(String[] args) {
    String s1 = " W ";
    putO(s1);
    s1.concat("W");
    System.out.println("|" + s1 + "|");
  }
}
```

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

```
String s = "This is hard";
s = "-" + s.substring(s.indexOf(' '), s.indexOf(' ', s.indexOf(' ') + 1) + 1)
    .strip() + "-";
System.out.println(s);
```

```
String txt = """
  a
```

```
    b
  c
""";
int from = 0;
int to = txt.indexOf('\n');
String line = null;
while(to < txt.length()-1) {
  to = txt.indexOf('\n', from);
  line = txt.substring(from, to);
  System.out.print(line.length());
  from = to+1;
}
```

```
public class MyClass {
  public static void main(String[] args) {
    String str = "hello";
    StringBuilder sb = new StringBuilder(str);
    sb.reverse();
    if (str == sb) System.out.println("a");
    if (str.equals(sb)) System.out.println("b");
    if (sb.equals(str)) System.out.println("c");
  }
}
```

```
public class MyClass {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("Have a nice day");
    sb.setLength(6);
    System.out.println(sb);
  }
}
```

```
public class RQ_8_24 {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("  1234  ");
    sb.trimToSize();
    sb.append("!");
    sb.reverse();
    sb.setLength(5);
    System.out.println("|" + sb + "|");
  }
}
```

```
public class PeskyCharSeq {
  public static void main (String[] args) {
    StringBuilder sb1 = new StringBuilder("WOW");
    StringBuilder sb2 = new StringBuilder(sb1);
    System.out.println((sb1==sb2) + " " + sb1.equals(sb2));
  }
}
```

```
public class MoreCharSeq {
  public static void main (String[] args) {
    String s1 = "WOW";
    StringBuilder s2 = new StringBuilder(s1);
    String s3 = new String(s2);
    System.out.println((s1.hashCode() == s2.hashCode()) + " " +
                       (s1.hashCode() == s3.hashCode()));
  }
}
```

```
public class Appendage {
  private static void putO(StringBuilder s1) {
    s1.append("O");
  }

  public static void main(String[] args) {
    StringBuilder s1 = new StringBuilder("W");
    putO(s1);
    s1.append("W!");
    System.out.println(s1);
  }
}
```

```
StringBuilder text = new StringBuilder();
text.append("12");
text.insert(1, "34");
text.delete(1, 1);
text.replace(0, 1, "");
System.out.println(text);
```

```
StringBuilder text = new StringBuilder();
text.append("42");
text.delete(1,2);
System.out.println(text.toString() + (text.capacity() + text.length()));
```

```
static int    abs(int i)
static long   abs(long l)
static float  abs(float f)
static double abs(double d)
static int    min(int a, int b)
static long   min(long a, long b)
static float  min(float a, float b)
static double min(double a, double b)
static int    max(int a, int b)
static long   max(long a, long b)
static float  max(float a, float b)
static double max(double a, double b)
```

```
long   ll = Math.abs(2022L);              // 2022L
double dd = Math.abs(-Math.PI);           // 3.141592653589793

double d1 = Math.min(Math.PI, Math.E);    // 2.718281828459045
long   m1 = Math.max(1984L, 2022L);       // 2022L
int    i1 = (int) Math.max(3.0, 4);       // Cast required.
```

```
static double ceil(double d)
static double floor(double d)
static int  round(float f)
static long round(double d)
```

```
double upPI    = Math.ceil(3.14);               // 4.0
double downPI  = Math.floor(3.14);              // 3.0
long   roundPI = Math.round(3.14);              // 3L

double upNegPI    = Math.ceil(-3.14);           // -3.0
double downNegPI  = Math.floor(-3.14);          // -4.0
long   roundNegPI = Math.round(-3.14);          // -3L
```

```
static double pow(double d1, double d2)
static double exp(double d)
```

static double log(double d)

```
static double sqrt(double d)
```

```
double r = Math.pow(2.0, 4.0);              // 16.0
double v = Math.exp(2.0);                   // 7.38905609893065
double l = Math.log(Math.E);                // 0.9999999999999981
double c = Math.sqrt(25.0);                 // 5.0
```

```
out.println(Integer.MAX_VALUE + 1);                   // (1a) -2147483648
out.println(Math.addExact(Integer.MAX_VALUE, 1));     // (1b) ArithmeticException
out.println(Math.addExact(1_000_000, 1_000));         // (1c) 1001000

out.println(Integer.MAX_VALUE * 100);                 // (2a) -100
out.println(
     Math.multiplyExact(Integer.MAX_VALUE, 100)       // (2b) ArithmeticException
);
out.println(Math.multiplyExact(1_000_000, 1_000));    // (2c) 1000000000


out.println((int)Long.MAX_VALUE);                     // (3a) -1
out.println(Math.toIntExact(Long.MAX_VALUE));         // (3b) ArithmeticException
out.println(Math.toIntExact(1_000_000));              // (3c) 1000000
```

```
static int  absExact(int a)
static long absExact(long a)
static int  addExact(int x, int y)
static long addExact(long x, long y)
static int  subtractExact(int x, int y)
static long subtractExact(long x, long y)
static int  multiplyExact(int x, int y)
static long multiplyExact(long x, int y)
static long multiplyExact(long x, long y)
static int  incrementExact(int a)
static long incrementExact(long a)
static int  decrementExact(int a)
static long decrementExact(long a)
static int  negateExact(int a)
static long negateExact(long a)
static int toIntExact(long value)
```

```
int diceValue = 1 + (int)(Math.random() * 6.0);  // A dice roll in range [1 .. 6]
```

`Math.random()`: `{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}`
Multiply by 6: `{0.0, 0.6, 1.2, 1.8, 2.4, 3.0, 3,6, 4.2, 4.8, 5.4}`
Convert to `int`: `{0,   0,   1,   1,   2,   3,   3,   4,   4,   5}`
Add offset 1: `{1,   1,   2,   2,   3,   4,   4,   5,   5,   6}`

```
int     nextInt()
int     nextInt(int bound)
long    nextLong()
float   nextFloat()
double  nextDouble()
boolean nextBoolean()
```

```
Random generator = new Random();
```

```
int number = generator.nextInt();
```

```
number = generator.nextInt(11);      // Random integer in the interval [0, 10]
number = 2 + generator.nextInt(11);  // Random integer in the interval [2, 12]
number = 2 + 3*generator.nextInt(5); // Random integer in the set {2, 5, 8, 11, 14}
```

```
int diceValue = 1 + generator.nextInt(6);  // A dice value in the interval [1, 6]
```

```
Random persistentGenerator = new Random(31);
```

```
The BigDecimal  3.14   has unscaled value =  314   and scale = 2.
The BigDecimal -3.1415 has unscaled value = -31415 and scale = 4.
```

`BigDecimal(String strValue)`

```
static BigDecimal valueOf(long value)
static BigDecimal valueOf(double value)
```

```
BigDecimal dTobd   = new BigDecimal(0.7);         // 0.6999999999999999555910...
BigDecimal strTobd = new BigDecimal("0.7");       // 0.7
BigDecimal valTobd = BigDecimal.valueOf(0.7);     // 0.7
```

```
double d1 = 0.70;
double d2 = 0.10;
System.out.println(d1 + d2);
```

Output: `0.7999999999999999`

```
BigDecimal bd1b = new BigDecimal("0.70");
BigDecimal bd2b = new BigDecimal("0.10");
System.out.println(bd1b.add(bd2b));
```

Output: `0.80`

```
BigDecimal price     = new BigDecimal("2.99");
BigDecimal tax       = new BigDecimal("0.25");
BigDecimal quantity  = BigDecimal.TEN.pow(3);
BigDecimal totalCost = price.add(price.multiply(tax)).multiply(quantity);
System.out.println(totalCost);
```

Output: `3737.5000`

```
BigDecimal sum = BigDecimal.ZERO;
sum.add(BigDecimal.ONE);           // sum is not updated! Returns new BigDecimal.
```

```
BigDecimal add(BigDecimal val)
BigDecimal subtract(BigDecimal val)
BigDecimal multiply(BigDecimal val)
BigDecimal divide(BigDecimal val)
BigDecimal remainder(BigDecimal val)
```

```
class MyClass {
    public static void main(String[] args) {
        double v = 10.5;

        System.out.println(Math.ceil(v));        // (1)
        System.out.println(Math.round(v));       // (2)
        System.out.println(Math.floor(v));       // (3)
        System.out.println((int) Math.ceil(v));  // (4)
        System.out.println((int) Math.floor(v)); // (5)
    }
}
```

```
public class Round {
    public static void main(String[] args) {
        System.out.println(Math.round(-0.5) + " " + Math.round(0.5));
    }
};
```

```
class TLC {                          // (1) Top-level class

  // Static member types:
  static class     SMC {}            // (2) Static member class
         interface SMI {}            // (3) Static member interface
         enum      SME {}            // (4) Static member enum
         record    SMR() {}          // (5) Static member record

  // Non-static member class:
  class NSMC {}                      // (6) Inner class

  // Local types in non-static context (analogous for static context):
  void nsm() {                       // Non-static method
    class     LC {}                  // (7) Local class (inner class)
    interface SLI {}                 // (8) Static local interface
    enum      SLE {}                 // (9) Static local enum
    record    SLR() {}               // (10) Static local record
  }

  // Anonymous classes (here defined as initializer expressions):
         SMC nsf = new SMC() {};  // (11) Inner class in non-static context
  static SMI  sf = new SMI() {};  // (12) Inner class in static context
}
```

```
// File: ListPool.java
package smc;

public class ListPool {                            // (1) Top-level class

  public static class MyLinkedList {               // (2) Static member class

    private interface ILink { }                    // (3) Static member interface

    public static class BiNode                     // (4) Static member class
                  implements IBiLink {
```

```
      public static void printSimpleName() {     // (5) Static method
        System.out.println(BiNode.class.getSimpleName());
      }

      public void printName() {                  // (6) Instance method
        System.out.println(this.getClass().getName());
      }
    } // end BiNode
  } // end MyLinkedList

  interface IBiLink
            extends MyLinkedList.ILink {         // (7) Static member interface
//  private static class Traversal { }           // (8) Compile-time error!
                                                 //     Can only be public.
    class BiTraversal { }                        // (9) Class is public and static
  } // end IBiLink

  public class SortedList {                      // (10) Non-static member class
    private static class SortCriteria {}         // (11) Static member class
  }
}
```

```
// File: MyBiLinkedList.java
package smc;

public class MyBiLinkedList implements ListPool.IBiLink {       // (12)

  public static void main(String[] args) {
    ListPool.MyLinkedList.BiNode.printSimpleName();             // (13)
    ListPool.MyLinkedList.BiNode node1
             = new ListPool.MyLinkedList.BiNode();              // (14)
    node1.printName();                                          // (15)

//  ListPool.MyLinkedList.ILink ref;                  // (16) Compile-time error!
  }
}
```

```
BiNode
smc.ListPool$MyLinkedList$BiNode
```

```
ListPool$IBiLink$BiTraversal.class
ListPool$IBiLink.class
ListPool$MyLinkedList$BiNode.class
ListPool$MyLinkedList$ILink.class
ListPool$MyLinkedList.class
ListPool$SortedList.class
ListPool.class
```

```
    ListPool.MyLinkedList.BiNode objRef1
            = new ListPool.MyLinkedList.BiNode();               // (14)
```

```
// File: Client1.java
import smc.ListPool.MyLinkedList;                          // (1) Type import

public class Client1 {
  MyLinkedList.BiNode objRef1 = new MyLinkedList.BiNode(); // (2)
}
```

```
// File: Client2.java
import static smc.ListPool.MyLinkedList.BiNode;         // (3) Static import

public class Client2 {
  BiNode objRef2 = new BiNode();                         // (4)
}

//class BiListPool implements smc.ListPool.IBiLink { }  // (5) Compile-time error!
                                                        // Not accessible!
// File: ListPool.java
public class ListPool {                                 // Top-level class

  public void messageInListPool() {                     // Instance method
    System.out.println("This is a ListPool object.");
  }

  private static class MyLinkedList {                   // (1) Static member class
    private static int maxNumOfLists = 100;             // Static field
    private int currentNumOfLists;                      // Instance field

    public static void messageInLinkedList() {          // Static method
      System.out.println("This is MyLinkedList class.");
    }

    interface ILink { int MAX_NUM_OF_NODES = 2000; }// (2) Static member interface

    protected static class Node implements ILink {  // (3) Static member class
```

```
      private int max = MAX_NUM_OF_NODES;           // (4) Instance field

      public void messageInNode() {                 // Instance method
//      int currentLists = currentNumOfLists;// (5) Not OK. Access instance field
                                             //             in outer class
        int maxLists = maxNumOfLists;     // Access static field in outer class
        int maxNodes = max;               // Access instance field in member class

//      messageInListPool();    // (6) Not OK. Call instance method in outer class
        messageInLinkedList();  // (7) Call static method in outer class
      }

      public static void main(String[] args) {
        int maxLists = maxNumOfLists;    // (8) Access static field in outer class
//      int maxNodes = max;   // (9) Not OK. Access instance field in member class
        messageInLinkedList();// (10) Call static method in outer class
      }
    }  // Node
  }  // MyLinkedList
} // ListPool
```

```
>javac ListPool.java
>java ListPool$MyLinkedList$Node
This is MyLinkedList class.
```

```
int maxLists = ListPool.MyLinkedList.maxNumOfLists;              // (8')
ListPool.MyLinkedList.messageInLinkedList();                     // (10')
```

```
>java ListPool$MyLinkedList$Node
```

```
// File: ListClient.java
class MyLinkedList {                                  // (1)
  private String message = "Shine the light";       // (2)

  public Node makeNode(String info, Node next) {    // (3)
    return new Node(info, next);                    // (4)
  }

  public class Node {                               // (5) Non-static member class
    // Static field:
    static int maxNumOfNodes = 100;                 // (6)
    // Instance fields:
    private String nodeInfo;                        // (7)
    private Node next;

    // Non-zero argument constructor:
    public Node(String nodeInfo, Node next) {       // (8)
      this.nodeInfo = nodeInfo;
      this.next = next;
    }

    // Instance methods:
    public Node getNext() { return next; }
    @Override
    public String toString() {
      return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")"; // (9)
    }
  }
}
//______________________________________________________________________________
public class ListClient {                                   // (10)
  public static void main(String[] args) {                  // (11)
    MyLinkedList list = new MyLinkedList();                 // (12)
    MyLinkedList.Node node1 = list.makeNode("node1", null); // (13)
    MyLinkedList.Node node2 = list.new Node("node2", node1);// (14)
    for (MyLinkedList.Node node = node2;
         node!=null;
         node = node.getNext()) {                           // (15)
      System.out.println(node);
    }

//  MyLinkedList.Node nodeX
//               = new MyLinkedList.Node("nodeX", node1);   // (16) Not OK.

  }
}
```

```
Shine the light in node1 (100)
Shine the light in node2 (100)
```

enclosing_object_reference`.new` *non_static_member_class_constructor_call*

```
    return new Node(info, next);                    // (4)
```

```
return this.new Node(info, next);                     // (4')
```

```
MyLinkedList.Node node2 = list.new Node("node2", node1);          // (14)
```

```
public static Node makeNode(String info, Node next) {       // (3') Static method
  return new MyLinkedList().new Node(info, next);   // (4') Explicit outer object
}
```

```
return this.message + " in " + this.nodeInfo +
                       " (" + this.maxNumOfNodes + ")";          // (9a) Not ok.
```

```
return MyLinkedList.this.message + " in " + this.nodeInfo +
                                     " (" + this.maxNumOfNodes + ")"; // (9b)
```

```
// File: OuterInstances.java
class TLClass {                                            // (1)  TLC
  private String id = "TLClass ";                          // (2)
  public TLClass(String objId) { id = id + objId; }        // (3)
  public void printId() {                                  // (4)
    System.out.println(id);
  }

  class InnerB {                                           // (5)  NSMC
    private String id = "InnerB ";                         // (6)
    public InnerB(String objId) { id = id + objId; }       // (7)
    public void printId() {                                // (8)
      System.out.print(TLClass.this.id + " : ");           // (9)  Refers to (2)
      System.out.println(id);                              // (10) Refers to (6)
    }

    class InnerC {                                         // (11) NSMC
      private String id = "InnerC ";                       // (12)
      public InnerC(String objId) { id = id + objId; }     // (13)
      public void printId() {                              // (14)
        System.out.print(TLClass.this.id + " : ");         // (15) Refers to (2)
        System.out.print(InnerB.this.id + " : ");          // (16) Refers to (6)
        System.out.println(id);                            // (17) Refers to (12)
      }
      public void printIndividualIds() {                   // (18)
        TLClass.this.printId();                            // (19) Calls (4)
        InnerB.this.printId();                             // (20) Calls (8)
        printId();                                         // (21) Calls (14)
      }
    } // InnerC
  } // InnerB
} // TLClass
//_____________________________________________________________________________
public class OuterInstances {                                // (22)
  public static void main(String[] args) {                   // (23)
    TLClass a = new TLClass("a");                            // (24)
    TLClass.InnerB b = a.new InnerB("b");                    // (25) b  --> a
    TLClass.InnerB.InnerC c1 = b.new InnerC("c1");           // (26) c1 --> b
    TLClass.InnerB.InnerC c2 = b.new InnerC("c2");           // (27) c2 --> b
    b.printId();                                             // (28)
    c1.printId();                                            // (29)
    c2.printId();                                            // (30)
    System.out.println("------------");
    TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
    TLClass.InnerB.InnerC cc = bb.new InnerC("cc");          // (32)
    bb.printId();                                            // (33)
    cc.printId();                                            // (34)
    System.out.println("------------");

    TLClass.InnerB.InnerC ccc =
      new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
    ccc.printId();                                           // (36)
    System.out.println("------------");

    ccc.printIndividualIds();                                // (37)
  }
}
```

```
TLClass a : InnerB b
TLClass a : InnerB b : InnerC c1
TLClass a : InnerB b : InnerC c2
------------
TLClass aa : InnerB bb
TLClass aa : InnerB bb : InnerC cc
------------
TLClass aaa : InnerB bbb : InnerC ccc
------------
TLClass aaa
TLClass aaa : InnerB bbb
TLClass aaa : InnerB bbb : InnerC ccc
```

```
      System.out.println(id);                          // (17)
      System.out.println(this.id);                     // (17a)
      System.out.println(InnerC.this.id);              // (17b)
```

```
TLClass a = new TLClass("a");                         // (24)
TLClass.InnerB b = a.new InnerB("b");                 // (25) b  --> a
TLClass.InnerB.InnerC c1 = b.new InnerC("c1");        // (26) c1 --> b
```

```
TLClass a : InnerB b : InnerC c1
```

```
TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");  // (31)
...
TLClass.InnerB.InnerC ccc =
  new TLClass("aaa").new InnerB("bbb").new InnerC("ccc");// (35)
```

```
// File: HiddenAndInheritedAccess.java
class Superclass {
  protected int value = 3;                 // (1) Instance field in superclass
}
//_____________________________________________________________________________
class TopLevelClass {                       // (2) Top-level Class
  private double value = 3.14;              // (3) Hidden by the instance field
                                            //     at (1) in the inner subclass
  class InnerSubclass extends Superclass {  // (4) Non-static member subclass
    public void printHidden() {             // (5)
      // (6) value from superclass:
      System.out.println("this.value: " + this.value);

      // (7) value from enclosing context:
      System.out.println("TopLevelClass.this.value: "
                         + TopLevelClass.this.value);
    }
  } // InnerSubclass
} // TopLevelClass
//____________________________________________________________________________
public class HiddenAndInheritedAccess {
  public static void main(String[] args) {
    TopLevelClass.InnerSubclass ref = new TopLevelClass().new InnerSubclass();
    ref.printHidden();
  }
}
```

```
this.value: 3
TopLevelClass.this.value: 3.14
```

```
// File: Extending.java
class OuterA {                                  // (1)
  class InnerA { }                              // (2)
}
//_____________________________________________________________________________
class SubInnerA extends OuterA.InnerA {         // (3) Extends NSMC at (2)
```

```
  // (4) Mandatory non-zero argument constructor:
  SubInnerA(OuterA outerRef) {
    outerRef.super();                             // (5) Explicit super() call
  }
}
//_______________________________________________________________________________
class OuterB extends OuterA {                     // (6) Extends class at (1)
  class InnerB extends OuterA.InnerA { }          // (7) Extends NSMC at (2)
}
//_______________________________________________________________________________
public class Extending {
  public static void main(String[] args) {

    // (8) Outer instance passed explicitly in constructor call:
    SubInnerA obj1 = new SubInnerA(new OuterA());
    System.out.println(obj1.getClass());

    // (9) No outer instance passed explicitly in constructor call to InnerB:
    OuterB.InnerB obj2 = new OuterB().new InnerB();
    System.out.println(obj2.getClass());
  }
}
```

```
public class MyClass {
  public static void main(String[] args) {
    Outer objRef = new Outer();
    System.out.println(objRef.createInner().getSecret());
  }
}

class Outer {
  private int secret;
  Outer() { secret = 123; }

  class Inner {
    int getSecret() { return secret; }
  }

  Inner createInner() { return new Inner(); }
}
```

```
public class Nesting {
  public static void main(String[] args) {
    B.C obj = new B().new C();
  }
}

class A {
  int val;
  A(int v) { val = v; }
}

class B extends A {
  int val = 1;
  B() { super(2); }

  class C extends A {
    int val = 3;
    C() {
      super(4);
      System.out.println(B.this.val);
      System.out.println(C.this.val);
      System.out.println(super.val);
    }
  }
}
```

```
public class Outer {
  public void doIt() {}
  public class Inner {
    public void doIt() {}
  }

  public static void main(String[] args) {
    new Outer().new Inner().doIt();
  }
}
```

```
// File: LocalClient1.java
class TLCWithSLClass {                              // Top-level Class

  static void staticMethod(final int fp) {         // (1) Static Method
//  StaticLocal slRef = new StaticLocal(10);       // (2) Class cannot be resolved
//  System.out.println(StaticLocal.staticValue()); // (3) Class cannot be resolved


//  public static class StaticLocal { // (4) Not OK. Cannot be static,
                                      //             and no access modifier
    final class StaticLocal {                        // (5) Static local class
      public static final int sf1 = 10;              // (6) Static field
      private int if1;                               // (7) Instance field
      public StaticLocal(int val) {                  // (8) Constructor
        this.if1 = val;
      }
      public int getValue() { return if1; }          // (9) Instance method
      public static int staticValue() { return sf1; }// (10) Static method
    } // end StaticLocal

    StaticLocal slRef2 = new StaticLocal(100);                          // (11)
    System.out.println("Instance field: " + slRef2.if1);                // (12)
    System.out.println("Instance method call: " + slRef2.getValue());   // (13)
    System.out.println("Static method call: " + StaticLocal.staticValue());// (14)
  } // end staticMethod
}

public class LocalClient1 {
  public static void main(String[] args) {
    TLCWithSLClass.staticMethod(100);                                   // (15)
  }
}
```

```
// File: LocalClient2.java
class TLCWithNSLClass {                // Top-level Class

  void nonStaticMethod(final int fp) { // (1) Non-static Method
    // Local variables:
    int flv = 10;              // (2) Effectively final local variable
    final int hlv = 20;        // (3) Final local variable (constant variable)
    int nflv1 = 30;            // (4) Non-final local variable
    int nflv3;                 // (5) Non-final local variable declaration

    nflv1 = 40;                // (6) Not effectively final local variable

    // Non-static local class
    class NonStaticLocal {// (7)
      int f1 = fp;         // (8) Final param from enclosing method
      int f2 = flv;        // (9) Effectively final variable from enclosing method
//    int f3 = nflv1;      // (10) Not effectively final from enclosing method
//    int f4 = nflv2;      // (11) Name nflv2 cannot be resolved: use-before-decl
//    int f5 = nflv3;      // (12) Not definitely assigned
      int hlv;             // (13) Shadows local variable at (3)
      NonStaticLocal (int value) {
        hlv = value;
       System.out.println("Instance field: " + hlv);// (14) Prints value from (13)
      }
    } // end NonStaticLocal

    NonStaticLocal nslRef = new NonStaticLocal(200);// (15) Implicit outer object
    int nflv2 = 50;                                 // (16) Attempted use in (11)
    nflv3 = 60;                                     // (17) Initializes (4)
    System.out.println("Local variable: " + hlv);   // (18) Prints value from (3)
  } // end nonStaticMethod
}

public class LocalClient2 {
  public static void main(String[] args) {
    new TLCWithNSLClass().nonStaticMethod(1000);    // (19)
  }
}
```

```
    double f4 = TLCWith2LCS.this.nsf1;      // (5) In enclosing object.
```

```
// File: LocalClient3.java
class Base { protected int nsf1; }     // Superclass
//_____________________________________________________________________________
class TLCWith2LCS {                    // Top-level Class
  private int nsf1;                    // Non-static field
  private int nsf2;                    // Non-static field
  private static int sf;               // Static field

  void nonStaticMethod( int fp) {      // Non-static Method

    class NonStaticLocal extends Base {// (1) Non-static local subclass
      int f1 = nsf1;                   // (2) Inherited from superclass.
      int f2 = this.nsf1;              // (3) Inherited from superclass.
      int f3 = super.nsf1;             // (4) Inherited from superclass.
      int f4 = TLCWith2LCS.this.nsf1;  // (5) In enclosing object.
      int f5 = nsf2;                   // (6) Instance field in enclosing object.
      int f6 = sf;                     // (7) static field from enclosing class.
    } // NonStaticLocal

  } // nonStaticMethod

  static void staticMethod(final int fp) { // Static Method

    class StaticLocal extends Base {   // (8) Static local subclass
      int f1 = nsf1;                   // (9) Inherited from superclass.
      int f2 = this.nsf1;              // (10) Inherited from superclass.
      int f3 = super.nsf1;             // (11) Inherited from superclass.
//    int f4 = TLCWith2LCS.this.nsf1;  // (12) No enclosing object.
//    int f5 = nsf2;                   // (13) No enclosing object.
      int f6 = sf;                     // (14) static field from enclosing class.
    } // StaticLocal

  } // staticMethod
}

public class LocalClient3 {
  public static void main(String[] args) {
    TLCWith2LCS.staticMethod(200);          // (15)
    new TLCWith2LCS().nonStaticMethod(100); // (16)
  }
}
```

```
// File: LocalClassClient.java
interface IDrawable {                     // (1)
  void draw();
}
//_____________________________________________________________________________
class Shape implements IDrawable {        // (2)
  @Override
  public void draw() { System.out.println("Drawing a Shape."); }
}
//_____________________________________________________________________________
class Painter {                           // (3) Top-level Class
  public Shape createCircle(final double radius) { // (4) Non-static Method
    class Circle extends Shape {          // (5) Non-static local class
```

```
      @Override
      public void draw() {
        System.out.println("Drawing a Circle of radius: " + radius); // (6)
      }
    }
    return new Circle();                  // (7) Passed enclosing object reference
  }

  public static IDrawable createGraph() { // (8) Static Method
    class Graph implements IDrawable {    // (9) Static local class
      @Override
      public void draw() { System.out.println("Drawing a Graph."); }
    }
    return new Graph();                   // (10) Object of static local class
  }
}
//_____________________________________________________________________________
public class LocalClassClient {
  public static void main(String[] args) {
    IDrawable[] drawables = {             // (11)
        new Painter().createCircle(5),    // (12) Object of non-static local class
        Painter.createGraph(),            // (13) Object of static local class
        new Painter().createGraph()       // (14) Object of static local class
    };
    for (IDrawable aDrawable : drawables) // (15)
      aDrawable.draw();

    System.out.println("Local Class Names:");
    System.out.println(drawables[0].getClass().getName());   // (16)
    System.out.println(drawables[1].getClass().getName());   // (17)
  }
}
```

```
Drawing a Circle of radius: 5.0
Drawing a Graph.
Drawing a Graph.
Local Class Names:
Painter$1$Circle
Painter$1$Graph
```

```
// File: LocalTypesClient.java
class LocalTypes {                        // Top-level Class
  private int nsf;                        // (1) Non-static field
  private static int sf;                  // (2) Static field

  void nonStaticMethod(final int fp) {    // (3) Non-static Method. Final parameter.
    int lv = 20;                          // (4) Local variable

    record StaticLocalRecord(int val) { // Static local record
      // Cannot access local variables:
//    static int f1 = fp;  // (5) Cannot access final param from enclosing method.
//    static int f2 = lv;  // (6) Cannot access effectively final local variable
                           //     from enclosing method.

      // Can only access static fields in enclosing context:
      static int f3 = sf;         // (7) Access static field in enclosing context.

      void printFieldsFromEnclosingContext() {
//      System.out.println(nsf);  // (8) Cannot access non-static field
                                  //     in enclosing context.
        System.out.println(sf);   // (9) Access static field in enclosing context.
      }
    }

    // (10) Create local record. No enclosing instance passed to the constructor.
    StaticLocalRecord lrRef = new StaticLocalRecord(100);
    System.out.println("Value: " + lrRef.val());
  } // nonStaticMethod
}

public class LocalTypesClient {
  public static void main(String[] args) {
    new LocalTypes().nonStaticMethod(1000);    // (10)
  }
}
```

```
new superclass_name<optional_type_arguments> (optional_constructor_arguments)
{
 member_declarations
}
```

```
// File: AnonClassClient.java
interface IDrawable {                             // (1)
  void draw();
}
//_______________________________________________________________________________
class Shape implements IDrawable {                // (2)
  @Override
  public void draw() { System.out.println("Drawing a Shape."); }
}
//_______________________________________________________________________________
class Painter {                                   // (3) Top-level Class

  public Shape createShape() {                    // (4) Non-static Method
    return new Shape() {                          // (5) Extends superclass at (2)
      @Override
      public void draw() { System.out.println("Drawing a new Shape."); }
    };
  }
  public static IDrawable createIDrawable() {    // (7) Static Method
    return new IDrawable() {                      // (8) Implements interface at (1)
      @Override
      public void draw() {
        System.out.println("Drawing a new IDrawable.");
      }
    };
  }
}
//_______________________________________________________________________________
public class AnonClassClient {
  public static void main(String[] args) {       // (9)
    IDrawable[] drawables = {                     // (10)
        new Painter().createShape(),              // (11) Non-static anonymous class
        Painter.createIDrawable(),                // (12) Static anonymous class
        new Painter().createIDrawable()           // (13) Static anonymous class
    };
    for (IDrawable aDrawable : drawables)         // (14)
      aDrawable.draw();

    System.out.println("Anonymous Class Names:");
    System.out.println(drawables[0].getClass().getName());// (15)
    System.out.println(drawables[1].getClass().getName());// (16)
  }
}
```

```
// ...
class Shape implements IDrawable {          // (2)
  @Override public void draw() { System.out.println("Drawing a Shape."); }
}

class Painter {                             // (3) Top-level Class

  public Shape createShape() {              // (4) Non-static Method
    return new Shape() {                    // (5) Extends superclass at (2)
      @Override public void draw() { System.out.println("Drawing a new Shape."); }
    };
  }
  // ...

}
// ...
```

```
new interface_name<optional_parameterized_types>() { member_declarations }
```

```
interface IDrawable {                          // (1) Interface
  void draw();
}
// ...
class Painter {                                // (3) Top-level Class
  // ...
  public static IDrawable createIDrawable() {  // (7) Static Method
    return new IDrawable() {                   // (8) Implements interface at (1)
      @Override public void draw() {
        System.out.println("Drawing a new IDrawable.");
      }
```

```
        };
    }
}
// ...
```

```
quitButton.addActionListener(
    new ActionListener() {    // (1) Anonymous class implements an interface
      // Invoked when the user clicks the quit button.
      @Override public void actionPerformed(ActionEvent evt) {
        System.exit(0);       // (2) Terminates the program
      }
    }
);
```

```
// File: AnonClient.java
abstract class Base {           // (1) Superclass
  protected int nsf1;
  abstract void printValue();
}
//_____________________________________________________________________________
class TLCWithAnonClasses {      // (2) Top level Class
  private int nsf1;             // Non-static field
  private int nsf2;             // Non-static field
  private static int sf = 5;    // Static field

  public void nonStaticMethod(final int fp) { // (3) Non-static Method
    // Local variables:
    int flv = 10;               // (4) Effectively final local variable
    final int hlv = 20;         // (5) Final local variable (constant variable)
    int nflv1 = 30;             // (6) Non-final local variable
    int nflv3;                  // (7) Non-final local variable declaration

    nflv1 = 40;                 // (8) Not effectively final local variable

    Base baseRef = new Base() { // (9) Non-static anonymous class
      // Static fields: Accessing local declarations in the enclosing block:
      static int sff1 = fp;   // (10) Final param from enclosing method
      static int sff2 = flv;  // (11) Effect. final variable from enclosing method
//    static int sf1 = nflv1; // (12) Not effect. final from enclosing method

      // Instance fields: Accessing local declarations in the enclosing block:
      int f1 = fp;       // (13) Final param from enclosing method
      int f2 = flv;      // (14) Effectively final variable from enclosing method
//    int f3 = nflv1;    // (15) Not effectively final from enclosing method

//    int f4 = nflv2;    // (16) nflv2 cannot be resolved: not decl-before-use
//    int f5 = nflv3;    // (17) Not definitely assigned: not initialized
      int hlv;           // (18) Shadows local variable at (5)

      // Accessing member declarations inherited from superclass:
      int f6 = nsf1;                   // (19) Inherited from superclass
      int f7 = this.nsf1;              // (20) Inherited from superclass
      int f8 = super.nsf1;             // (21) Inherited from superclass

      // Accessing (hidden) member declarations in the enclosing class:
      int f9 = TLCWithAnonClasses.this.nsf1;          // (22) In enclosing object
      int f10 = nsf2;                  // (23) Instance field in enclosing object
      int f11 = sf;                    // (24) Static field from enclosing class

      { nsf1 = fp; }                   // (25) Non-static initializer block

      @Override void printValue() {                        // (26) Instance method
        System.out.println("Instance field nsf1: " + nsf1);// (27)
      }
    };
    int nflv2 = 70;
    nflv3 = 80;
    baseRef.printValue();              // (28) Invoke method on anonymous object
  }

  public static final Base baseField = new Base() { // (29) Static anonymous class
    // Accessing (hidden) member declarations in the enclosing class:
//  int f1 = TLCWithAnonClasses.this.nsf1; // (30) Not OK. No enclosing object
//  int f2 = nsf2;                         // (31) Not OK. No enclosing object
    { nsf1 = sf; }                         // (32) Non-static initializer block

    @Override void printValue() {                         // (33) Instance method
      System.out.println("Instance field nsf1: " + nsf1);// (34)
    }
  };
}
//_____________________________________________________________________________
public class AnonClient {
  public static void main(String[] args) {
    new TLCWithAnonClasses().nonStaticMethod(100);                 // (35)
    TLCWithAnonClasses.baseField.printValue();                     // (36)
  }
}
```

```
Instance field nsf1: 100
Instance field nsf1: 5
```

```
interface IntHolder { int getInt(); }
```

```
//----(1)----
  IntHolder makeIntHolder(int i) {
    i = 10;
    return new IntHolder() {
      public int getInt() { return i; }
    };
  }
```

```
//----(2)----
  IntHolder makeIntHolder(final int i) {
    return new IntHolder {
      public int getInt() { return i; }
    };
  }
//----(3)----
  IntHolder makeIntHolder(int i) {
    class MyIH implements IntHolder {
      public int getInt() { return i; }
    }
    return new MyIH();
  }
//----(4)----
  IntHolder makeIntHolder(final int i) {
    class MyIH implements IntHolder {
      public int getInt() { return i; }
    }
    return new MyIH();
  }
//----(5)----
  IntHolder makeIntHolder(int i) {
    return new MyIH(i);
  }
  static class MyIH implements IntHolder {
    final int j;
    MyIH(int i) { j = i; }
    public int getInt() { return j; }
  }
```

```
public class Access {
  final String str1 = "Access.str1";

  public static void main(final String args[]) {
    final String str1 = "LocalVar.str1";

    class Helper { String getStr1() { return str1; } }
    class Inner {
      String str1 = "Inner.str1";
      Inner() {
        System.out.println( /* (1) INSERT EXPRESSION HERE */ );
      }
    }
    Inner inner = new Inner();
  }
}
```

```
public class Test {
  private char x = '=';
  public static void main(String[] args) {
    char x = '<';
    Test t = new Test() {
      private char x = '>';
      public String toString() {
        return this.x + super.toString() + x;
      }
    };
    System.out.println(t);
  }
  public String toString() {
    return x + "42";
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    int x = 42;
    String s = new String() {
      int x = 24;
      public int hashCode() {
        return this.x;
      }
    };
    System.out.println(s.hashCode());
  }
}
```

```
import java.io.*;

class WellBehavedClass {
  // ...
  void wellBehavedMethod() {
    FileReader reader;
    long[] bigArray = new long[20000];
    // ... uses local variables ...
    // Does clean-up (before starting something extensive)
    reader = null;                    // (1)
    bigArray = null;                  // (2)

    // Start some other extensive activity
    // ...
  }
  // ...
}
```

```
public class MyClass {
  public static void main(String[] args) {
    String msg;
    String pre = "This program was called with ";
    String post = " as first argument.";
    String arg1 = new String((args.length > 0) ? "'" + args[0] + "'" :
                                "<no argument>");
    msg = arg1;
    arg1 = null;             // (1)
    msg = pre + msg + post;  // (2)
    pre = null;              // (3)
    System.out.println(msg);
    msg = null;              // (4)
    post = null;             // (5)
    args = null;             // (6)
  }
}
```

```
public class Elements {
  public static void main(String[] args) {
    int[] array = new int[4];
    for (int i = 0; i < 4; i++) {
      array[i] = i;
    }
    array[0] = array[1] = array[2] = array[3] = 0;
    System.gc();                                      // (1);
  }
}
```

```
public class Link {
  private Link next;
  Link(Link next) { this.next = next; }

  public static void main(String[] args) {
    Link p = null;
    for (int i = 0; i < 5; i++) {
      p = new Link(p);
    }
    System.gc();                                    // (1)
  }
}
```

```
public class Album {
  private Song[] songs = { new Song(), new Song() };
  public Song[] getSongs() {
    return songs;
  }
  public void removeAll() {
    songs = null;
  }
}
```

```
public class Test {
```

```
public class Test {
  public static void main(String[] args) {
    Album album = new Album();
    Song[] songs = album.getSongs();
    album.removeAll();
    System.out.println(album.getSongs());
    // Line (1)
  }
}
```

```
class ConstantInitializers {
        int minAge = 12;             // (1) Non-static
```

```
  static double pensionPoints = 10.5; // (2) Static
  // ...
}
```

```
class MoreInitializers {
         int noOfDays    = 7 * NO_OF_WEEKS;    // (1) Non-static
  static int NO_OF_WEEKS = 52;                 // (2) Static
  // ...
}
```

```
class NonStaticInitializers {
  int length  = 10;                    // (1)
//double area = length * width;        // (2) Not OK. Illegal forward reference.
  double area = length * this.width;  // (3) OK, but width has default value 0.
  int width   = 10;                    // (4)

  int sqSide = height = 20;            // (5) OK. Legal forward reference.
  int height;                          // (6)
}
```

```
// File: TestOrder.java
class Hotel {
  private int noOfRooms        = 12;                                   // (1)
  private int maxNoOfGuests    = initMaxGuests();                      // (2) Bug
  private int occupancyPerRoom = 2;                                    // (3)
  public int initMaxGuests() {                                   // (4)
    System.out.println("occupancyPerRoom: " + occupancyPerRoom);
    System.out.println("maxNoOfGuests:    " + noOfRooms * occupancyPerRoom);
    return noOfRooms * occupancyPerRoom;
  }

  public int getMaxGuests() { return maxNoOfGuests; }            // (5)

  public int getOccupancy() { return occupancyPerRoom; }         // (6)
}
//________________________________________________________________________
public class TestOrder {
  public static void main(String[] args) {
    Hotel hotel = new Hotel();                                   // (7)
    System.out.println("AFTER OBJECT CREATION");
    System.out.println("occupancyPerRoom: " + hotel.getOccupancy()); // (8)
    System.out.println("maxNoOfGuests:    " + hotel.getMaxGuests()); // (9)
  }
}
```

```
// File: ExceptionsInInitializers.java
class RoomOccupancyTooHighException
      extends RuntimeException {}                     // (1) Unchecked Exception
class TooManyHotelsException
      extends Exception {}                            // (2) Checked Exception
//_____________________________________________________________________________
class Hotel {
  // Static Members
  private static int noOfHotels = 12;
  private static Hotel[] hotelPool = createHotelPool();   // (3)

  private static Hotel[] createHotelPool() {              // (4)
    try {
      if (noOfHotels > 10)
        throw new TooManyHotelsException();
    } catch (TooManyHotelsException e) {
      noOfHotels = 10;
      System.out.println("No. of hotels adjusted to " + noOfHotels);
    }
    return new Hotel[noOfHotels];
  }
  // Instance Members
  private int noOfRooms        = 215;
  private int occupancyPerRoom = 5;
  private int maxNoOfGuests    = initMaxGuests();         // (5)

  private int initMaxGuests() {                           // (6)
    if (occupancyPerRoom > 4)
      throw new RoomOccupancyTooHighException();
    return noOfRooms * occupancyPerRoom;
  }
}
//_____________________________________________________________________________
public class ExceptionsInInitializers {
  public static void main(String[] args) {
    try { new Hotel(); }
    catch (RoomOccupancyTooHighException exception) {
      exception.printStackTrace();
    }
  }
}
```

```
No. of hotels adjusted to 10
RoomOccupancyTooHighException
      at Hotel.initMaxGuests(ExceptionsInInitializers.java:29)
      at Hotel.<init>(ExceptionsInInitializers.java:25)
      at ExceptionsInInitializers.main(ExceptionsInInitializers.java:36)
```

```
class MatrixData {

  static final int ROWS = 12, COLUMNS = 10;          // (1)
  static long[][] matrix = new long[ROWS][COLUMNS];  // (2)
  // ...
  static {                                           // (3) Static initializer
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = 2*i + j;
  }
  // ...
}
```

```
public class ScheduleV1 {
//static int numOfDays  = 7 * numOfWeeks; // (1) Compile-time error! Simple name.
  static int numOfWeeks = 52;             // (2)
}
```

```
public class ScheduleV2 {
  static int numOfWeeks = 52;                              // (2) 52
  static int numOfDays  = 7 * numOfWeeks;                  // (1) 364
}
```

```
public class ScheduleV3 {
  static int numOfDays  = 7 * ScheduleV3.numOfWeeks;  // (1) 0
  static int numOfWeeks = 52;                          // (2) 52
}
```

```
public class ScheduleV3 {
  static int numOfDays;                                   // Default value: 0
  static int numOfWeeks;                                  // Default value: 0
  static {
    numOfDays  = 7 * numOfWeeks;                          // (1) 0
    numOfWeeks = 52;                                      // (2) 52
  }
}
```

```
public class ScheduleV4 {
  static int numOfDays  = 7 * ScheduleV4.numOfWeeks;  // (1) 364
  static final int numOfWeeks = 52;                    // (2) final: 52
}
```

```
public class ScheduleV4 {
  static final int numOfWeeks = 52;                    // Constant variable: 52
  static int numOfDays;                                // Default value: 0
  static {
    numOfDays  = 7 * numOfWeeks;                       // (1) 364
  }
}
```

```
package refs1;

public class ForwardRefs {

  static {                    // (1) Static initializer block
    System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 1: sf1=0, sf2=0

    sf1 = 10;                 // (2) OK. Assignment to sf1 allowed
//  sf1 = if1;           // (3) Not OK. Non-static field access in static context
//  int a = 2 * sf1;     // (4) Not OK. Read operation before declaration
    var b = sf1 = 20;         // (5) OK. Assignment to sf1 allowed
    int c = ForwardRefs.sf1;  // (6) OK. Not accessed by simple name

    System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 1:  sf1=20, sf2=0
  }

  // Field declarations:
  static int sf1 = sf2 = 30;  // (7) Static field. Assignment to sf2 allowed
  static int sf2;              // (8) Static field
  int if1 = 5;                 // (9) Non-static field

  static {                     // (10) Static initializer block
    System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Enter static block 2: sf1=30, sf2=30

    int d = 2 * sf1;           // (11) OK. Read operation after declaration
    var e = sf1 = 50;          // (12) OK. Assignment to sf1 allowed

    System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
        ForwardRefs.sf1, ForwardRefs.sf2); // Exit static block 2:  sf1=50, sf2=30
  }

  public static void main(String[] args) {
  }
}
```

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

```
package refs2;

public class ForwardRefsSimulated {

  // Declaration of static fields:
  static int sf1;              // (1) Initialized to default value: 0
  static int sf2;              // (2) Initialized to default value: 0

  static {                     // (3) Static initializer block
    // Code from static block 1:
```

```
    System.out.printf("Enter static block 1: sf1=%s, sf2=%s%n",
        sf1, sf2);            // Enter static block 1: sf1=0, sf2=0

    sf1 = 10;                 // (4) sf1 gets the value 10
    var b = sf1 = 20;         // (5) b and sf1 get the value 20
    int c = sf1;              // (6) c gets the value 20

    System.out.printf("Exit static block 1:  sf1=%s, sf2=%s%n",
        sf1, sf2);            // Exit static block 1:  sf1=20, sf2=0

    // Initializer expressions for field declaration:
    sf1 = sf2 = 30;           // (7) sf1 and sf2 get the value 30

    // Code from static block 2:
    System.out.printf("Enter static block 2: sf1=%s, sf2=%s%n",
        sf1, sf2);            // Enter static block 2: sf1=30, sf2=30

    int d = 2 * sf1;          // (8) d gets the value 60
    var e = sf1 = 50;         // (9) e and sf1 get the value 50

    System.out.printf("Exit static block 2:  sf1=%s, sf2=%s%n",
        sf1, sf2);            // Exit static block 2:  sf1=50, sf2=30
  }

  public static void main(String[] args) {
  }
}
```

```
Enter static block 1: sf1=0, sf2=0
Exit static block 1:  sf1=20, sf2=0
Enter static block 2: sf1=30, sf2=30
Exit static block 2:  sf1=50, sf2=30
```

```
// File: ExceptionInStaticInitBlocks.java
package version1;

class TooManyCellsException extends Exception {          // (1) Checked Exception
  TooManyCellsException(String number) { super(number); }
}
```

```
//______________________________________________________________________________
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells;                 // (2) No initializer expression

  static {                                    // (3) Static block
    try {                                     // (4) Handles checked exception
      if (noOfCells > 300)
        throw new TooManyCellsException(String.valueOf(noOfCells));
    } catch (TooManyCellsException e) {
      System.out.println("Exception handled: " + e);
      noOfCells = 300;
      System.out.println("No. of cells adjusted to " + noOfCells);
    }
    cells = new int[noOfCells];
  }
}
//______________________________________________________________________________
public class ExceptionInStaticInitBlocks {
  public static void main(String[] args) {
    new Prison();
  }
}
```

```
Exception handled: version1.TooManyCellsException: 365
No. of cells adjusted to 300
```

```
class Prison {
  // Static Members
  private static int   noOfCells = 365;
  private static int[] cells = initPrison();  // (2) Initializer expression

  //
  private static int[] initPrison() {         // (3) Private static method
    try {                                     // (4) Handles checked exception
      if (noOfCells > 300)
        throw new TooManyCellsException(String.valueOf(noOfCells));
    } catch (TooManyCellsException e) {
      System.out.println("Exception handled: " + e);
      noOfCells = 300;
      System.out.println("No. of cells adjusted to " + noOfCells);
    }
    return new int[noOfCells];
  }
}
```

```
class InstanceInitializers {

  long[] squares = new long[10];    // (1)
  // ...
  {                                 // (2) Instance Initializer
    for (int i = 0; i < squares.length; i++)
      squares[i] = i*i;
  }
  // ...
}
```

```
public class InstanceInitializersII {

  { //Instance initializer with forward references.   (1)
    i = j = 10;                                      // (2) Permitted.
    int result = this.i * this.j;                    // (3) i is 10, j is 10.
    System.out.println(this.i);                    // (4) 10
    System.out.println(this.j);                    // (5) 10
    System.out.println(this.k);                    // (6) 50
  }
  // Instance field declarations.
  int i;                 // (7) Field declaration without initializer expression.
  int j = 100;           // (8) Field declaration with initializer expression.
  final int k = 50;      // (9) Final instance field with constant expression.

}
```

```
public class NonStaticForwardReferences {

  {                           // (1) Instance initializer block.
    nsf1 = 10;                // (2) OK. Assignment to nsf1 allowed.
    nsf1 = sf1;               // (3) OK. Static field access in non-static context.
    //  int a = 2 * nsf1;  // (4) Not OK. Read operation before declaration.
    var b = nsf1 = 20;        // (5) OK. Assignment to nsf1 allowed.
    int c = this.nsf1;        // (6) OK. Not accessed by simple name.
  }

  int nsf1 = nsf2 = 30;       // (7) Non-static field. Assignment to nsf2 allowed.
  int nsf2;                   // (8) Non-static field.
  static int sf1 = 5;         // (9) Static field.

  {                           // (10) Instance initializer block.
    int d = 2 * nsf1;         // (11) OK. Read operation after declaration.
    var e = nsf1 = 50;        // (12) OK. Assignment to nsf1 allowed.
  }

  public static void main(String[] args) {
    NonStaticForwardReferences objRef = new NonStaticForwardReferences();
    System.out.println("nsf1: " + objRef.nsf1);
    System.out.println("nsf2: " + objRef.nsf2);
  }
}
```

```
// File: InstanceInitBlock.java
class Base {
  protected int a;
  protected int b;
  void print() { System.out.println("a: " + a); }
}
//______________________________________________________________________________
class AnonymousClassMaker {
  Base createAnonymous() {
    return new Base() {           // (1) Anonymous class
      {                           // (2) Instance initializer
        a = 5; b = 10;
      }

      @Override
      void print() {
        super.print();
        System.out.println("b: " + b);
      }
    };  // end anonymous class
  }
}
//______________________________________________________________________________
public class InstanceInitBlock {
  public static void main(String[] args) {
    new AnonymousClassMaker().createAnonymous().print();
  }
}
```

```
// File: ExceptionsInInstBlocks.java
class RoomOccupancyTooHighException
      extends Exception {}                    // (1) Checked exception
class BankruptcyException
      extends RuntimeException {}             // (2) Unchecked exception
//_______________________________________________________________________________
class Hotel {
  // Instance Members
  private boolean bankrupt         = true;
  private int     noOfRooms        = 215;
  private int     occupancyPerRoom = 5;
  private int     maxNoOfGuests;

  {                                           // (3) Instance initializer block
    try {                                     // (4) Handles checked exception
      if (occupancyPerRoom > 4)
        throw new RoomOccupancyTooHighException();
    } catch (RoomOccupancyTooHighException exception) {
      System.out.println("ROOM OCCUPANCY TOO HIGH: " + occupancyPerRoom);
      occupancyPerRoom = 4;
    }
    maxNoOfGuests = noOfRooms * occupancyPerRoom;
  }

  {                                           // (5) Instance initializer block
    if (bankrupt)
      throw new BankruptcyException();         // (6) Throws unchecked exception
  }    // ...
}
//_______________________________________________________________________________
public class ExceptionsInInstBlocks {
  public static void main(String[] args) {
    new Hotel();
  }
}
```

```
ROOM OCCUPANCY TOO HIGH: 5
Exception in thread "main" BankruptcyException
        at Hotel.<init>(ExceptionsInInstBlocks.java:27)
        at ExceptionsInInstBlocks.main(ExceptionsInInstBlocks.java:33)
```

```java
// File: ObjectConstruction.java
class SuperclassA {
  public SuperclassA() {                     // (1) Superclass constructor
    System.out.println("Constructor in SuperclassA");
  }
}
//_______________________________________________________________________________
class SubclassB extends SuperclassA {

  SubclassB() {                              // (2) No-argument constructor
    this(3);
    System.out.println("No-argument constructor in SubclassB");
  }

  SubclassB(int i) {                         // (3) Non-zero argument constructor
    System.out.println("Non-zero argument constructor in SubclassB");
    value = i;
  }

  {                                          // (4) Instance initializer block
    System.out.println("Instance initializer block in SubclassB");
    value = 2;                               // (5)
  }

  int value = initializerExpression();       // (6) Instance field declaration

  private int initializerExpression() {      // (7)
    System.out.println("Instance initializer expression in SubclassB");
    return 1;
  }
}
//_______________________________________________________________________________
public class ObjectConstruction {
  public static void main(String[] args) {
    SubclassB objRef = new SubclassB();         // (8)
    System.out.println("value: " + objRef.value);
  }
}
```

```
Constructor in SuperclassA
Instance initializer block in SubclassB
Instance initializer expression in SubclassB
Non-zero argument constructor in SubclassB
No-argument constructor in SubclassB
value: 3
```

```
// File: ObjectInitialization.java
class SuperclassA {
  protected int superValue;                              // (1)
  SuperclassA() {                                        // (2)
    System.out.println("Constructor in SuperclassA");
    this.doValue();                                      // (3)
  }
  void doValue() {                                       // (4)
    this.superValue = 911;
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
}
//_______________________________________________________________________________
class SubclassB extends SuperclassA {
  private int value = 800;                               // (5)
  SubclassB() {                                          // (6)
    System.out.println("Constructor in SubclassB");
    this.doValue();
    System.out.println("superValue (from SuperclassA): " + this.superValue);
  }
  @Override
  void doValue() {                                       // (7)
    System.out.println("value (from SubclassB): " + this.value);
  }
}
//_______________________________________________________________________________
public class ObjectInitialization {
  public static void main(String[] args) {
    System.out.println("Creating an object of SubclassB.");
    new SubclassB();                                     // (8)
  }
}
```

```
Creating an object of SubclassB.
Constructor in SuperclassA
value (from SubclassB): 0
Constructor in SubclassB
value (from SubclassB): 800
superValue (from SuperclassA): 0
```

```
public class MyClass {
  private static int count = 5;
  static final int STEP = 10;
  boolean alive;

  // (1) INSERT STATIC INITIALIZER BLOCK HERE
}
```

```
public class MyClass {
  public static void main(String[] args) {
    MyClass obj = new MyClass(n);
  }
  static int i = 5;
  static int n;
  int j = 7;
  int k;

  public MyClass(int m) {
    System.out.println(i + ", " + j + ", " + k + ", " + n + ", " + m);
  }

  { j = 70; n = 20; } // Instance initializer block

  static { i = 50; }  // Static initializer block
}
```

```
public class FirstClass {
  static int gap = 10;
  double length;
  final boolean active;

  // (1) INSERT CODE HERE
}
```

```
public class Initialization {
  private static String msg(String msg) {
    System.out.println(msg);
    return msg;
  }

  public Initialization() { m = msg("1"); }

  { m = msg("2"); }
  String m = msg("3");

  public static void main(String[] args) {
    Object obj = new Initialization();
  }
}
```

```
public class Music {
  static {
    System.out.print("-C-");
  }
  {
    System.out.print("-D-");
  }
  public Music(){
    System.out.print("-E-");
  }
}
```

```
public class Song extends Music {
  static {
    System.out.print("-F-");
  }
  {
    System.out.print("-G-");
  }
  public Song(){
    System.out.print("-A-");
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Music x1 = new Song();
    Song x2 = new Song();
  }
}
```

```
class GeomInit {
//int width = 14;                     // (1)

  {
//  area = width * height;            // (2)
  }

  int width = 37;

  {
//  height = 11;                      // (3)
  }

  int height, area;
//area = width * height;              // (4)

  {
//  int width = 15;                   // (5)
    area = 100;
  }
}
```

```
class LegacyNode {
  private Object    data;    // The value in the node
  private LegacyNode next;   // The reference to the next node.
  LegacyNode(Object data, LegacyNode next) {
    this.data = data;
    this.next = next;
  }
  public void       setData(Object obj)      { this.data = obj; }
  public Object     getData()                { return this.data; }
  public void       setNext(LegacyNode next) { this.next = next; }
  public LegacyNode getNext()                { return this.next; }
  @Override public String toString() {
    return this.data + (this.next == null? "" : ", " + this.next);
  }
}
```

```
LegacyNode node1 = new LegacyNode(4, null);       // 4 --> null
LegacyNode node2 = new LegacyNode("July", node1); // "July" --> 4 --> null
```

```
Object obj = node2.getData();
```

```
if (obj instanceof String str) {
  System.out.println(str.toUpperCase()); // Method specified in the String class.
}
```

```
class Node<E> {
  private E            data;    // Data                              (1)
  private Node<E>      next;    // Reference to next node            (2)
  Node(E data, Node<E> next) {                                    // (3)
    this.data = data;
    this.next = next;
  }
  public void    setData(E data)       { this.data = data; }   // (4)
  public E       getData()             { return this.data; }   // (5)
  public void    setNext(Node<E> next) { this.next = next; }   // (6)
  public Node<E> getNext()             { return this.next; }   // (7)
  @Override public String toString() {                          // (8)
    return this.data.toString() +
           (this.next == null ? "" : ", " + this.next.toString());
  }
}
```

```
class Node<E> {
  ...
  Node<E>() { ... }                                        // Compile-time error!
  ...
}
```

```
E ref = new E();                           // Compile-time error!
```

```
class Node<E> {
  private static E e1;                                  // (1) Compile-time error!
  public  static E oneStaticMethod(E e2) {              // (2) Compile-time error!
    E e3;                                               // (3) Compile-time error!
    System.out.println(e3);
  }
  // ...
}
```

```
Node<Integer> intNode = new Node<Integer>(2020, null);
```

```
Integer iRef = intNode.getData();           // Integer object with int value 2020
```

```
intNode.setData(2020);                              // Ok.
intNode.setData("TwentyTwenty");                    // (1) Compile-time error!
intNode.setNext(new Node<Integer>(2019, null));     // (2020, (2019, null))
intNode.setNext(new Node<String>("Hi", null));      // (2) Compile-time error!
```

```
Node<String> strNode = new Node<String>("Hi", null);
intNode = strNode;                          // (3) Compile-time error!
String str = strNode.getData();             // (4) No explicit cast necessary.
```

```
Node<String> lst = new Node<String>("Hi", null); // Explicit actual type parameter
```

```
Node<String> lst = new Node<>("Hi", null); // Actual type parameter inferred.
```

```
new Node<>(null, null);  // Actual type parameter: Object.
```

```
Node<String>  strNode = new Node<>(null, null); // Actual type parameter: String.
Node<Integer> intNode = new Node<>(null, null); // Actual type parameter: Integer.
Node<Number>  numNode = new Node<>(null, null); // Actual type parameter: Number.
Node<Number>  lstNode = new Node<>(2021, null); // Actual type parameter: Number.
```

```
// (1) Method declaration with parameterized type as formal parameter.
void find(Node<String> node) { /* ... */ }
...
// (2) Method call where actual argument uses diamond operator.
find(new Node<>(null,null));  // Actual type parameter: Object or String?
```

```
HashMap<String, List<Integer>> map = new HashMap<>();
HashMap<String, List<Integer>> map = new HashMap<String,<>>(); // Error!
```

```
Node<String> rawNode = new Node("Hi", null); // Unchecked conversion warning!
```

```
new Node<>("Hi", null) {/* ... */}; // Parameterized type: Node<String>
new Node("Hi", null)   {/* ... */}; // Raw type: Node
```

```
// Parameterized local variable declarations:
Node<String> node1 = new Node<String>(null, null); // (1) Node of String
Node<String> node2 = new Node<>(null, null);       // (2) Node of String
var node3 = new Node<String>(null, null);          // (3) Node of String
var node4 = new Node<>(null, null);                // (4) Node of Object
```

```
interface IMonoLink<E> {
  void         setData(E data);
  E            getData();
  void         setNext(IMonoLink<E> next);
  IMonoLink<E> getNext();
}

class MonoNode<E> implements IMonoLink<E> {
  private E            data;    // Data
  private IMonoLink<E> next;    // Reference to next node                    (1)

  MonoNode(E data, IMonoLink<E> next) {                                   // (2)
    this.data = data;
    this.next = next;
  }

  @Override public void setData(E data) { this.data = data; }
  @Override public E    getData()       { return this.data; }
  @Override public void setNext(IMonoLink<E> next) { this.next = next; } // (3)
  @Override public IMonoLink<E> getNext()          { return this.next; } // (4)
  @Override public String toString() {
    return this.data.toString() + (this.next == null? "" : ", " + this.next);
  }
}
```

```
class MonoNode<E> implements IMonoLink<E> {
  // ...
}
```

```
IMonoLink<String> strNode2 = new MonoNode<>("Bye", null);
System.out.println(strNode2.getData());                      // Prints: Bye
```

```
IMonoLink<String> strNode3 = new IMonoLink<>("Bye", null); // Compile-time error!
```

```
// File: LymphNode.java
class Lymph { /*... */ }

public class LymphNode implements IMonoLink<Lymph> {
  private Lymph            body;
  private IMonoLink<Lymph> location;
  @Override public void  setData(Lymph obj) { body = obj; }
  @Override public Lymph getData()          { return body; }
  @Override public void  setNext(IMonoLink<Lymph> loc) { this.location = loc; }
  @Override public IMonoLink<Lymph> getNext()          { return this.location; }
}
```

```
interface IBiLink<E> extends IMonoLink<E> {
  // ...
}
```

```
class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  // ...
}
```

```
BiNode<Integer> intBiNode = new BiNode<>(2020, null, null);
MonoNode<Integer> intMonoNode = intBiNode;          // (1)
Integer iRef = intMonoNode.getData();               // Integer with int value 2020
MonoNode<Number> numMonoNode = intBiNode;           // (2) Compile-time error!
```

```
interface IBiLink<T> extends IMonoLink<T> {
  void       setPrevious(IBiLink<T> previous);
  IBiLink<T> getPrevious();
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E>  previous;    // Reference to previous node

  BiNode(E data, IBiLink<E> next, IBiLink<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  @Override public void setPrevious(IBiLink<E> previous) {
    this.previous = previous;
  }
  @Override public IBiLink<E> getPrevious() { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
            this.getData() +
           (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

```
class AbstractNode { /* ... */ }                       // A non-generic supertype
class SimpleNode<E> extends AbstractNode { /* ... */ }// A generic subtype
class IntegerBiNode extends BiNode<Integer> {         // A non-generic subtype
  IntegerBiNode(Integer data, IntegerBiNode next, IntegerBiNode previous) {
    super(data, next, previous);
  }
  //...
}
```

```
class WeirdNode<E> extends MonoNode<E> implements IMonoLink<Integer> { // Error!
  //...
}
```

```
...
Node<Integer> intNode = new Node<>(2020, null);
Integer iRef = intNode.getData();        // Integer object with int value 2020
...
Node rawNode = intNode;           // (5) Assigning to raw type always possible.
rawNode.setData("BOOM");          // (6) Unchecked call warning!
intNode = rawNode;                // (7) Unchecked conversion warning!
iRef = intNode.getData();         // (8) ClassCastException!
iRef = rawNode.getData();         // (9) Compile-time error!
```

```
// A client for the generic class Node<E> in Example 11.2, p. 568.
public class Preliminaries {
  public static void main(String[] args) {
    Node<Integer> intNode = new Node<>(2018, null);
    Integer iRef = intNode.getData();        // Integer object with int value 2018
    intNode.setData(2020);                   // Ok.
//  intNode.setData("TwentyTwenty");         // (1) Compile-time error!
    intNode.setNext(new Node<>(2019, null)); // (2020, (2019, null))
//  intNode.setNext(new Node<>("Hi", null)); // (2) Compile-time error!

    Node<String> strNode = new Node<>("Hi", null);
//  intNode = strNode;               // (3) Compile-time error!
    String str = strNode.getData(); // (4) No explicit cast necessary.

    Node rawNode = intNode;          // (5) Assigning to raw type always possible.
    rawNode.setData("BOOM");         // (6) Unchecked call warning!
    intNode = rawNode;               // (7) Unchecked conversion warning!
    iRef = intNode.getData();        // (8) ClassCastException!
//  iRef = rawNode.getData();        // (9) Compile-time error!
  }
}
```

```
>javac -Xlint:unchecked Preliminaries.java
Preliminaries.java:16: warning: [unchecked] unchecked call to setData(E) as a
member of the raw type Node
    rawNode.setData("BOOM");        // (6) Unchecked call warning!
                   ^
  where E is a type-variable:
    E extends Object declared in class Node
Preliminaries.java:17: warning: [unchecked] unchecked conversion
    intNode = rawNode;              // (7) Unchecked conversion warning!
              ^
  required: Node<Integer>
  found:    Node
2 warnings
```

```
>java Preliminaries
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
 be cast to java.lang.Integer
        at Preliminaries.main(Preliminaries.java:18)
```

```
List wordList = new ArrayList();      // Using non-generic types.
wordList.add("two zero two zero");   // Can add any object.
wordList.add(2020);
//...
Object element = wordList.get(0);    // Always returns an Object.
//...
if (element instanceof String str) { // Runtime check to avoid ClassCastException.
  // Use reference str.
}
```

```
List<String> wordList = new ArrayList<>();         // Using a specific type.
wordList.add("two zero two zero");                 // Can add strings only.
wordList.add(2020);                                // Compile-time error!
//...
String element = wordList.get(0);                  // Always returns a String.
//...
```

```
Node<Integer> intNode    = new Node<>(2020,null);          // (1)
Node<Double>  doubleNode = new Node<>(3.14,null);          // (2)
Node<Number>  numNode    = new Node<>(2021, null);         // (3)
```

```
numNode.setData(10.5);                                    // (4)
numNode.setData(2022);                                    // (5)
numNode.setNext(intNode);                           // (6) Compile-time error!
numNode = new Node<Number>(2030, doubleNode);       // (7) Compile-time error!
numNode = intNode;                                    // (8) Compile-time error!
numNode = doubleNode;                                 // (9) Compile-time error!
numNode = intNode;                   // If this assignment was allowed,
numNode.setData(25.5);               // the data could be corrupted,
Integer iRef = intNode.getData();    // resulting in a ClassCastException!
```

```
Node<? extends Integer> intSubNode = new Node<Integer>(100, null);
Node<? extends Number> numSupNode = intSubNode;
```

```
Node<? super Number> numSupNode = new Node<Number>(100, null);
Node<? super Integer> numIntSupNode = numSupNode;
```

```
Node<?> anyNode = new Node<?>(2020, null);                 // Compile-time error!
Node<? extends Integer> extIntNodeA
               = new Node<? extends Integer>(0, null);     // Compile-time error!
Node<? extends Integer> extIntNodeB = new Node<Integer>(0, null);  // OK
```

```
class QuestionableNode<?>  { /* ... */ }                               // Not OK.
class SubNode extends Node<?> { /* ... */ }                             // Not OK.
interface INode extends Comparable<? extends Node<?>> { /* ... */ }  // Not OK.
class XNode implements Comparable<?>  { /* ... */ }                    // Not OK.
class OddNode extends Node<Node<?>> implements Comparable<Node<?>> { /* ... */ }
...
Node<?> nodeOfAnyNode = new Node<Node<?>>(new Node<Integer>(2020, null), null);
```

```
Node<Object>  objNode = new Node<Object>(100, null);
Node<Number>  numNode = new Node<Number>(200, null);
Node<Integer> intNode = new Node<Integer>(300, null);
Node<? extends Number> extNumNode
                       = intNode; // Node<Integer> <: Node<? extends Number>
Node<? super Integer>  supIntNode
                       = numNode; // Node<Number> <: Node<? super Integer>
supIntNode = objNode;             // Node<Object> <: Node<? super Integer>
numNode    = intNode;             // Compile-time error! Types unrelated.
```

```
Node<? super Number>   supNumNode;
Node<? extends Object> extObjNode;
Node<? super Object>   supObjNode;
extNumNode = supNumNode;  // (1) Compile-time error! Types unrelated.
extObjNode = supObjNode;  // (2) Node<? super Object> <: Node<? extends Object>
supObjNode = extObjNode;  // (3) Compile-time error!
```

```
Node    rawNode;
Node<?> anyNode;
rawNode = anyNode;  // Node <-- Node<?> is type-safe.
anyNode = rawNode;  // Node<?> <-- Node is type-safe.
```

```
// (4):
anyNode = extObjNode;  // Node<?> <-- Node<? extends Object> is type-safe.
extObjNode = anyNode;  // Node<? extends Object> <-- Node<?> is type-safe.

// (5):
anyNode    = rawNode;  // Node<?> <-- Node is type-safe.
extObjNode = rawNode;  // Node<? extends Object> <-- Node: Unchecked Conversion
```

```
extNumNode = rawNode; // Node<? extends Number> <-- Node: Unchecked Conversion
intNode    = rawNode; // Node<Integer> <-- Node: Unchecked Conversion
```

```
class Node<E> {
  private E data;
  // ...
  public void setData(E obj) { data = obj; }         // (1) Set operation.
  public E    getData()      { return data; }        // (2) Get operation.
  // ...
}
```

```
Node<Number> numNode = new Node<>(2020, null);
numNode.setData(2021);                      // (3) Can only set a Number.
Number data = numNode.getData();            // (4) Can only get a Number.
```

```
class WildcardAccessRestrictions {

  static void checkIt(Node s0) {                        // (h1)
//static void checkIt(Node<?> s0) {                     // (h2)
//static void checkIt(Node<? extends Number> s0) {  // (h3)
//static void checkIt(Node<? super Number> s0) {    // (h4)
//static void checkIt(Node<Number> s0) {                // (h5)
    // Local variables
    Object  object  = new Object();                     // (v1)
    Number  number  = 1.5;                              // (v2)
    Integer integer = 10;                               // (v3)
    // Method calls
    s0.setData(object);                                 // (s1)
    s0.setData(number);                                 // (s2)
    s0.setData(integer);                                // (s3)
    object  = s0.getData();                             // (s4)
    number  = s0.getData();                             // (s5)
    integer = s0.getData();                             // (s6)
    s0.setData(s0.getData());                           // (s7)
  }
}
```

```
void printNodeDataSequence(Node<?> s0) {        // Unbounded parameterized type
  Node<?> next = s0.getNext();                  // Returns node as Node<Object>
  while (next != null) {
    Object obj = next.getData();                // Object reference.
    System.out.println(obj.toString());         // Call Object method
    next = next.getNext();
  }
}
```

```
Long longInt = 20L;
Node<Long> longNode = new Node<>(longInt, null);  // Node of Long, that is
Node<? extends Number> s0 = longNode;// referenced by a Node<? extends Number> ref.
s0.setData(object);                    // If this was allowed, or             (1)
s0.setData(number);                    // if this was allowed, or             (2)
s0.setData(integer);                   // if this was allowed,                (3)
longInt = longNode.getData();          // we would get an exception here.     (4)
```

```
s0.setData(longInt);                    // Compile-time error!
```

```
Node<Number> numNode = new Node<>(2020, null);
Node<? super Number> s0 = numNode;
s0.setData(object);              // (1) If this set operation was allowed,
number = numNode.getData();      // (2) we would get an exception here.
```

```
Node<Object> objNode = new Node<>("Hi", null);  // String as data.
Node<? super Number> s0 = objNode;
number = s0.getData();        // (3) If allowed, we would get an exception here.
object = s0.getData();        // This is always ok.
```

```
class CmpNode<? extends Comparable> { ... }           // Compile-time error!
class CmpNode<E extends Comparable<E>> {              // E is bounded.
  // ...
}
```

```
class CmpNode<E extends Comparable<E>> extends Node<E> {
  // ...
}
```

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {
 // ...
}
```

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override public int compareTo(CmpNode<E> node2) {
    return this.getData().compareTo(node2.getData());
  }
}
```

```
CmpNode<Integer> intCmpNode  = new CmpNode<>(2020, null);         // (1)
CmpNode<Number>  numCmpNode  = new CmpNode<Number>(2020, null);   // (2) Error!
CmpNode<Integer> intCmpNode2 = new CmpNode<>(2021, null);
int result = intCmpNode.compareTo(intCmpNode2);
```

```
class CmpNode<E extends Number & Serializable> ...
class CmpNode<E extends Comparable<E> & Serializable> ...
class CmpNode<E extends Comparable<E> & Serializable & Comparable<String>> //Error
```

```
public class Utilities {

  // The key type and the array element type can be any type.
  static boolean containsV1(Object key, Object[] array) { // (1) Non-generic
                                                          //     version
    for (Object element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type and the array element type are the same.
  static <E> boolean containsV2(E key, E[] array) {       // (2) Generic version
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type is a subtype of the array element type.
  static <K extends E, E> boolean containsV3(K key, E[] array) {  // (3)
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
  }
}
```

```
static boolean containsV1(Object key, Object[] array) { // (1) Non-generic version
  // ...
}
```

```
static <E> boolean containsV2(E key, E[] array) {       // (2) Generic version
  // ...
}
```

```
public class ClassDecl {
  static <E_1,..., E_k> void genericMethod(P_1 p_1,..., P_m p_m) { ... }
  // ...
}
```

```
CallDecl ref;
ref.<A_1,..., A_k>genericMethod(a_1,..., a_m);
CallDecl.<A_1,..., A_k>genericMethod(a_1,..., a_m);
```

```
this.<A_1,..., A_k>genericMethod(a_1,..., a_m);            // Non-static context
super.<A_1,..., A_k>genericMethod(a_1,..., a_m);           // Non-static context
CallDecl.super.<A_1,..., A_k>genericMethod(a_1,..., a_m);  // Non-static context
```

```
<A_1,..., A_k>genericMethod(a_1,..., a_m);     // Compile-time error!
genericMethod(a_1,..., a_m);                   // Ok.
```

```
Integer[] intArray = {10, 20, 30};

boolean f1 = Utilities.<Integer>containsV2(20, intArray);           // (1) true
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f2 = Utilities.<Number>containsV2(30.5, intArray);          // (2) false
// E is Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f3 = Utilities.<Comparable<Integer>> containsV2(20, intArray); // (3) true
// E is Comparable<Integer>.
// Method signature:      containsV2(Comparable<Integer>, Comparable<Integer>[])
// Method call signature: containsV2(Integer,             Integer[])

boolean f4 = Utilities.<Integer>containsV2(30.5, intArray);         // (4) Error!
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Double,  Integer[])

// Requires explicit reference or raw type.
boolean f5 = <Integer>containsV2(20, intArray);              // (5) Syntax error!
boolean f6 = Utilities.containsV2(20, intArray);                  // (6) true
// E is inferred to be Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f7 = Utilities.containsV2(30.5, intArray);                // (7) false;
// E is inferred to be Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f8 = Utilities.containsV2("Hi", intArray);                // (8) false;
// E is inferred to be Serializable.
// Method signature:      containsV2(Serializable, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

```
boolean f9 = Utilities.<Integer>containsV2("Hi", intArray);         // (9) Error!
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(String, Integer[])
```

```
static <K extends E, E> boolean containsV3(K key, E[] array) {
  // ...
}
```

```
boolean f10 = Utilities.containsV3(30.5, intArray);              // (10) false
// K is inferred to be Double. E is inferred to be Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Double, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f11 = Utilities.containsV3("Hi", intArray);              // (11) false
// K is inferred to be String. E is inferred to be Serializable.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(String, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

```
boolean f12 = Utilities.<Number, Number>containsV3(30.0, intArray); // (12) false
// K is Number. E is Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Number, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f13 = Utilities.<Number, Integer>
                        containsV3(30.5, intArray);                 // (13) Error!
// K is Number. E is Integer.
// The constraint (K extends E) is not satisfied.

boolean f14 = Utilities.<Integer, Number>
                        containsV3(30.5, intArray);                 // (14) Error!
// K is Integer. E is Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Integer, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f15 = Utilities.<Number>containsV3(30.5, intArray);         // (15) Error!
// Incorrect no. of type parameters.
```

```
public static <K,V> Map<V,List<K>> toMultiMap(Map<K,V> origMap) { ... } // (16)
public static <N> Set<N> findVerticesOnPath(Map<N,Collection<N>> graph,
                                             N startVertex)     { ... } // (17)
```

```
/** Interface of a generic stack */
public interface IStack<E> extends Iterable<E> {
  void push(E element);           // Add the element to the top of the stack
  E pop();                        // Remove the element at the top of the stack.
  E peek();                       // Get the element at the top of the stack.
  int size();                     // No. of elements on the stack.
  boolean isEmpty();              // Determine if the stack is empty.
  boolean isMember(E element);    // Determine if the element is in the stack.
  E[] toArray(E[] toArray);       // Copy elements from stack to array
  @Override
  String toString();              // Return suitable text representation of
                                  // elements on the stack: (e1, e2, ..., en)
}
```

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/** Simplified implementation of a generic stack */
public class MyStack<E> implements IStack<E> {                     // (1)
  // Top of stack.
  private Node<E> tos;                                             // (2)
  // Size of stack
  private int numOfElements;                                       // (3)

  @Override public boolean isEmpty() { return tos == null; }       // (4)
  @Override public int size()        { return numOfElements; }     // (5)

  @Override public void push(E element) {                          // (6)
    tos = new Node<>(element, tos);
    ++numOfElements;
  }
  @Override public E pop() {                                          // (7)
    if (!isEmpty()) {
      E data = tos.getData();
      tos = tos.getNext();
      --numOfElements;
      return data;
    }
    throw new NoSuchElementException("No elements.");
  }

  @Override public E peek()  {                                        // (8)
    if (!isEmpty()) return tos.getData();
    throw new NoSuchElementException("No elements.");
  }

  // Membership
  @Override public boolean isMember(E element) {                      // (9)
    for (E data : this)
      if (data.equals(element))
        return true;       // Found.
    return false;          // Not found.
  }

  // Get iterator.
  @Override public Iterator<E> iterator() {                           // (10)
    return new NodeIterator<>(this.tos);
  }

  // Copy to array as many elements as possible.
  @Override public E[] toArray(E[] toArray) {                         // (11)
    Node<E> thisNode = tos;
    for (int i = 0; thisNode != null && i < toArray.length; i++) {
      toArray[i] = thisNode.getData();
      thisNode = thisNode.getNext();
    }
    return toArray;
  }

  // Text representation of stack: (e1, e2, ..., en).
  @Override public String toString() {                                // (12)
    StringBuilder rep = new StringBuilder("(");
    for (E data : this) {
      rep.append(data + ", ");
    }
    if (!isEmpty()) {
      int len = rep.length();
      rep.delete(len - 2, len);                      // Delete the last ", ".
    }
    rep.append(")");
    return rep.toString();
  }
}
```

```
import java.util.Iterator;

/** Iterator for nodes */
public class NodeIterator<E> implements Iterator<E> {
  private Node<E> thisNode;

  public NodeIterator(Node<E> first) { thisNode = first;  }

  @Override public boolean hasNext() { return thisNode != null; }

  @Override public E next() {
    E data = thisNode.getData();
    thisNode = thisNode.getNext();
    return data;
  }

  @Override public void remove() { throw new UnsupportedOperationException(); }
}
```

```
import java.util.*;

public class RQ100_50 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();     // (1)
    intList.add(2020);
    intList.add(2021);
    List<Number> numList = intList;                // (2)
    for (Number n : numList)                       // (3)
      System.out.println(n + " ");
  }
}
```

```
import java.util.*;

public class RQ100_40 {
  public static void main(String[] args) {
    List <? super Integer> sList = new ArrayList<Number>(); //(1)
    int i = 2020;
    sList.add(i);
    sList.add(++i);                                                 //(2)
    Number num = sList.get(0);                                      //(3)
  }
}
```

```
import java.util.*;

public class RQ100_70 {
  public static void main(String[] args) {
    List<Integer> glst1 = new ArrayList();          //(1)
    List nglst1 = glst1;                             //(2)
    List nglst2 = nglst1;                            //(3)
    List<Integer> glst2 = glst1;                     //(4)
  }
}
```

```
public class Box<T> {
  private T item;                                          // (1)

  private static T[] storage = new T[100];                 // (2)

  public Box(T item) { this.item = item; }                 // (3)

  public T getItem() { return item; }                      // (4)

  public void setItem(T newItem) { item = newItem; }   // (5)

  public static void getAllItems(T newItem) {              // (6)
    T temp;                                                // (7)
  }
}
```

```
Map<Integer, Map<Integer, String>> map1
              = new HashMap<Integer, HashMap<Integer, String>>();
```

```
Map<Integer, HashMap<Integer, String>> map2 = new HashMap<>();
```

```
Map<Integer, Integer> map3 = new HashMap<Integer, Integer>();
```

```
Map<? super Integer, ? super Integer> map4
              = new HashMap<? super Integer, ? super Integer>();
```

```
Map<? super Integer, ? super Integer> map5 = new HashMap<Number, Number>();
```

```
Map<? extends Number, ? extends Number> map6
              = new HashMap<Number, Number>();
```

```
Map <?,?> map7 = new HashMap<?,?>();
```

```
import java.util.*;

class Fruit {}
class Apple extends Fruit {}

public class RQ100_15 {
  public static void main(String[] args) {
    List<? extends Apple> lst1 = new ArrayList<Fruit>(); // (1)
    List<? extends Fruit> lst2 = new ArrayList<Apple>(); // (2)
    List<? super Apple> lst3 = new ArrayList<Fruit>();   // (3)
    List<? super Fruit> lst4 = new ArrayList<Apple>();   // (4)
    List<?> lst5 = lst1;                                 // (5)
    List<?> lst6 = lst3;                                 // (6)
    List lst7 = lst6;                                    // (7)
    List<?> lst8 = lst7;                                 // (8)
  }
}
```

```
import java.util.*;

public class RQ100_11 {
  public static void main(String[] args) {
    Set set = new TreeSet<String>();
    set.add("one");
    set.add(2);
    set.add("three");
    System.out.println(set);
  }
}
```

```
class Vehicle {}
class Car extends Vehicle {}
class Sedan extends Car {}

class Garage<V> {
  private V v;
  public V get() { return this.v; }
  public void put(V v) { this.v = v; }
}

public class GarageAdmin {

  private Object object = new Object();
  private Vehicle vehicle = new Vehicle();
  private Car car = new Car();
  private Sedan sedan = new Sedan();

  public void doA(Garage g) {
    g.put(object);              // (1)
    g.put(vehicle);             // (2)
    g.put(car);                 // (3)
    g.put(sedan);               // (4)
    object  = g.get();          // (5)
    vehicle = g.get();          // (6)
    car     = g.get();          // (7)
    sedan   = g.get();          // (8)
  }
}
```

```
import java.util.*;

public class GenParam {
  public static void main(String[] args) {
    List<Number> numList = new ArrayList<>();
    List<Integer> intList = new ArrayList<>();
    // (1) INSERT CODE HERE
  }

  static <T> void move(List<? extends T> lst1, List<? super T> lst2) {}
}
```

```
class Box<T> {
      Box()          {System.out.println(this);}                // (1)
  <V> Box(V v)       {System.out.println(v);}                   // (2)
  <V> Box(T t, V v) {System.out.println(t + ", " + v);}        // (3)
}
```

```
// File: WildcardCapture.java
...
Node<?>                 anyNode;
Node<? super Number>    supNumNode;

Node<Integer> intNode = anyNode;                 // (1) Compile-time error!
Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
anyNode.setData("Trash");                        // (3) Compile-time error!
```

```
WildcardCapture.java:10: error: incompatible types: Node<CAP#1> cannot be
converted to Node<Integer>
    Node<Integer> intNode = anyNode;                // (1) Compile-time error!
                            ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

```
WildcardCapture.java:11: error: incompatible types: Node<CAP#1> cannot be
converted to Node<? extends Number>
    Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
                                        ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object super: Number from capture of ? super Number
```

```
WildcardCapture.java:12: error: incompatible types: String cannot be converted to
CAP#1
    anyNode.setData("Trash");                       // (3) Compile-time error!
                    ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

```
public static <T> void move(MyStack<? extends T> srcStack,
                            MyStack<? super T> dstStack) {
  while (!srcStack.isEmpty())
    dstStack.push(srcStack.pop());
}
```

```
MyStack<?> anyStack;
MyStack.move(anyStack, anyStack);   // Compile-time error!
```

```
MyStackUser.java:68: error: method move in class MyStack<E#2> cannot be applied to
given types;
      MyStack.move(anyStack, anyStack);     // Compile-time error!
             ^
  required: MyStack<? extends E#1>,MyStack<? super E#1>
  found: MyStack<CAP#1>,MyStack<CAP#2>
  reason: cannot infer type-variable(s) E#1
    (argument mismatch; MyStack<CAP#2> cannot be converted to MyStack<? super E#1>)
  where E#1,E#2 are type-variables:
    E#1 extends Object declared in method <E#1>move(MyStack<? extends E#1>,My-
Stack<? super E#1>)
    E#2 extends Object declared in class MyStack
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Object from capture of ?
    CAP#2 extends Object from capture of ?
```

```
static void fillWithFirstV1(List<?> list) {
  Object firstElement = list.get(0);       // (1)
  for (int i = 1; i < list.size(); i++)
    list.set(i, firstElement);            // (2) Compile-time error
}
```

```
static <E> void fillWithFirstV2(List<E> list) {
  E firstElement = list.get(0);                // (3)
  for (int i = 1; i < list.size(); i++)
    list.set(i, firstElement);                 // (4)
}
```

```
static void fillWithFirstV3(List<?> list) {
  fillWithFirstV2(list);                    // (5) Type conversion
}
```

```
Collection<Number> colNum;
Set<Number> setNum;
Set<Integer> setInt;
colNum = setNum; // (1) Set<Number> <: Collection<Number>
colNum = setInt; // (2) Compile-time error!
```

```
Collection<Collection<Number>> colColNum; // Collection of Collections of Number
Set<Collection<Number>>        setColNum; // Set of Collections of Number
Set<Set<Integer>>              setSetInt; // Set of Sets of Integer
colColNum = setColNum;                    // (3) Set<Collection<Number>> <:
                                          //     Collection<Collection<Number>>
colColNum = setSetInt;                    // (4) Compile-time error!
setColNum = setSetInt;                    // (5) Compile-time error!
```

```
Collection<? extends Collection<Number>> colExtColNum;
colExtColNum = colColNum;       // (6) Collection<Collection<Number>> <:
                                //     Collection<? extends Collection<Number>>
colExtColNum = setColNum;       // (7) Set<Collection<Number>> <:
                                //     Collection<? extends Collection<Number>>
colExtColNum = setSetInt;       // (8) Compile-time error!
```

```
Collection<Collection<? extends Number>> colColExtNum;
colColExtNum = colColNum;       // (9)  Compile-time error!
colColExtNum = setColNum;       // (10) Compile-time error!
colColExtNum = setSetInt;       // (11) Compile-time error!
```

```
Map<Number, String>  mapNumStr;
Map<Integer, String> mapIntStr;
mapNumStr = mapIntStr;            // (12) Compile-time error!
```

```
Map<Integer, ? extends Collection<String>> mapIntExtColStr;
Map<Integer, Collection<? extends String>> mapIntColExtStr;
Map<Integer, Collection<String>>           mapIntColStr;
Map<Integer, Set<String>>                  mapIntSetStr;
mapIntExtColStr = mapIntColStr;// (13) Map<Integer, Collection<String>> <:
                               //      Map<Integer, ? extends Collection<String>>
mapIntExtColStr = mapIntSetStr;// (14) Map<Integer, Set<String>> <:
                               //      Map<Integer, ? extends Collection<String>>
```

```
mapIntColStr    = mapIntSetStr;    // (15) Compile-time error!
mapIntColExtStr = mapIntColStr;    // (16) Compile-time error!
mapIntColExtStr = mapIntSetStr;    // (17) Compile-time error!
```

```
public void moveFromV1(MyStack<E> srcStack) {             // (1)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());
}

public void moveFromV2(MyStack<? extends E> srcStack) {  // (2)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());
}

public void moveFromV3(MyStack<? super E> srcStack) {    // (3)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());                           // Compile-time error!
}
```

```
MyStack<Number> numStack   = new MyStack<>();                  // Stack of Number
numStack.push(5.5); numStack.push(10.5); numStack.push(20.5);
MyStack<Integer> intStack1 = new MyStack<>();                  // Stack of Integer
intStack1.push(5); intStack1.push(10); intStack1.push(20);
MyStack<Integer> intStack2 = new MyStack<>();                  // Stack of Integer
intStack2.push(15); intStack2.push(25); intStack2.push(35);
```

```
intStack1.moveFromV1(intStack2); // Ok.
numStack.moveFromV1(intStack2);  // Compile-time error!
```

```
intStack1.moveFromV2(intStack2);  // Pop from intStack2. Push on intStack1.
numStack.moveFromV2(intStack2);   // Pop from intStack2. Push on numStack.
```

```
public void moveToV1(MyStack<E> dstStack) {                  // (4)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}

public void moveToV2(MyStack<? extends E> dstStack) {        // (5)
  while (!this.isEmpty())
    dstStack.push(this.pop());                               // Compile-time error!
}

public void moveToV3(MyStack<? super E> dstStack) {          // (6)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}
```

```
intStack1.moveToV1(intStack2);    // Pop from intStack1. Push on intStack2.
intStack1.moveToV1(numStack);     // Compile-time error!

intStack1.moveToV3(intStack2);    // Pop from intStack1. Push on intStack2.
intStack1.moveToV3(numStack);     // Pop from intStack1. Push on numStack.
```

```
public static <E> void move(MyStack<? extends E> srcStack,  // (7)
                            MyStack<? super E> dstStack) {
  while (!srcStack.isEmpty())
    dstStack.push(srcStack.pop());
}

// Client code
MyStack.move(intStack1, intStack2);
MyStack.move(intStack1, numStack);
MyStack.move(numStack, intStack2);     // Compile-time error!
```

```
class MonoNode<E> implements IMonoLink<E> {
  private E                 data;    // Data
  private IMonoLink<E>      next;    // Reference to next node       (1)
  // ...
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E> previous;       // Reference to previous node   (2)
  // ...
}
```

```
public static <E> void traverseBinTree(IBiLink<E> root) {
  if (root.getPrevious() != null)
    traverseBinTree(root.getPrevious());
  System.out.print(root.getData() + ", ");
  if (root.getNext() != null)
    traverseBinTree((IBiLink<E>)root.getNext());  // (3) Cast necessary.
}
```

```
abstract class RecNode<E, T extends RecNode<E, T>>
abstract class RecNode<E, T extends RecNode<E, T>> {      // (1)
  private E data;                                         // (2)
  private T next;                                         // (3)

  RecNode(E data, T next) {
    this.data = data;
    this.next = next;
  }
  public void setData(E obj)  { data = obj; }
  public E    getData()       { return data; }
  public void setNext(T next) { this.next = next; }
  public T    getNext()       { return next; }
  @Override public String toString() {
    return this.data + (this.next == null ? "" : ", " + this.next);
  }
}
```

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>> {          // (4)

  private RecBiNode<E>  previous;    // Reference to previous node   // (5)

  RecBiNode(E data, RecBiNode<E> next, RecBiNode<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  public void setPrevious(RecBiNode<E> previous) { this.previous = previous; }
  public RecBiNode<E> getPrevious()              { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
        this.getData() + (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>>
```

```
public static <E> void traverseBinTree(RecBiNode<E> root) {     // (2)
  if (root.getPrevious() != null)
    traverseBinTree(root.getPrevious());
    System.out.print(root.getData() + ", ");
    if (root.getNext() != null)
      traverseBinTree(root.getNext());               // No cast necessary!
  }
```

```
// Pre-erasure code
List<String> strList = new ArrayList<>(); // (0)
List list = strList;       // (1) Assignment to non-generic reference is ok.
strList = list;            // (2) warning: unchecked conversion
strList.add("aha");        // (3) Method call type-safe.
list.add(23);              // (4) warning: [unchecked] unchecked call to add(E)
                           //     as a member of the raw type java.util.List
System.out.println(strList.get(1).length());  // (5) ClassCastException
```

```
// Post-erasure code
List strList = new ArrayList();                        // (0')
List list = strList;                                   // (1')
strList = list;                                        // (2')
strList.add("aha");                                    // (3')
list.add(Integer.valueOf(23));                         // (4')
System.out.println(((String)strList.get(1)).length()); // (5') Cast inserted.
```

```
class CmpNode extends Node implements Comparable {
  CmpNode(Object data, CmpNode next) {
    super(data, next);
  }
  public int compareTo(CmpNode node2) {       // (1) Implemented method erasure
    return this.getData().compareTo(node2.getData());
  }
  public int compareTo(Object node2) {        // (2) Inserted bridge method.
    return this.compareTo((CmpNode)node2);    // Calls the method at (1).
  }
}
```

```
static <T> void merge(MyStack<T> s1, MyStack<T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? super T> s2) { /*...*/ }
static <T> void merge(Node<T> s1, MyStack<T> s2)              { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, Node<? super T> s2)      { /*...*/ }
```

```
class MethodSGN<T> {
  void doIt(boolean b) {}                         // (1) void doIt(boolean)

  void doIt(T t) {}                               // (2) void doIt(Object)

  List<StringBuilder> doIt(StringBuilder sb) {   // (3) List doIt(StringBuilder)
    return null;
  }

  <E extends Comparable<E>> void doIt(E element) // (4) void doIt(Comparable)
    {}

  <E> E doIt(MyStack<? extends E> stack) {       // (5) Object doIt(MyStack)
    return null;
  }
}
```

```
void doIt(Object obj) {}                          // (2') void doIt(Object)

<E extends StringBuilder> List<E> doIt(E sb) {   // (3') List doIt(StringBuilder)
  return null;
}

void doIt(Comparable<T> element) {}              // (4') void doIt(Comparable)

<E> E doIt(MyStack<? super E> stack) {           // (5') Object doIt(MyStack)
  return null;
}
```

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override
  public boolean equals(CmpNode node2) {           // (1) Compile-time error.
//public boolean equals(Object node2) {            // (1') Correct header.
    return this.compareTo(node2) == 0;
  }

  @Override
  public int compareTo(Object node2) {              // (2) Compile-time error.
//public int compareTo(CmpNode<E> node2) {          // (2') Correct header
    return this.getData().compareTo(node2.getData());
  }
}
```

```
>javac CmpNode.java
...
CmpNode.java:8: method does not override or implement a method from a supertype
  @Override
  ^
...
CmpNode.java:14: method does not override or implement a method from a supertype
  @Override
  ^
```

```
import java.util.List;

class SupA {
  public void set(Integer ref)        {/*...*/}    // (1)
  public void set(List<Integer> list) {/*...*/}    // (2)
}

class SubA extends SupA {
  @Override public void set(Integer ref) {/*...*/}  // (1') same as at (1)
  @Override public void set(List list) {/*...*/}// (2') same as the erasure at (2)
}
```

```
public void set(Object t) {/*...*/}                  // (1)
public Object get()       {return null;}             // (2)
public void set(Object t) {set((Number)t);}        // (1')
```

```
class SupB<T> {
  public void set(T t) {/*...*/}                        // (1)
  public T get()       {return null;}                   // (2)
}
```

```
class SubB1 extends SupB<Number> {
  @Override public void set(Number num) {/*...*/} // (1a) Overrides
  @Override public Number get()       {return 0;} // (2a) Overrides
}

class SubB2 extends SupB<Number> {
  @Override public void set(Object obj) {/*...*/} // (1b) Error: same erasure
  @Override public void set(Long l)     {/*...*/} // (1c) Error: overloads

  @Override public Object get() {          // (2b) Error: incompatible return type
    return null;
  }
}
```

```
class SupZ<T> {
  public void set(T t) {/*...*/}              // (1)
}

class SubZ<E> extends SupZ {                  // Superclass not parameterized
  @Override public void set(E e) {/*...*/}  // (1a) Error: same erasure
}
```

```
class SupJ {
  public void set(Object obj) {/*...*/}          // (1)
  public Object get()    {return null;}          // (2)
}

class SubJ extends SupJ {
  @Override public <T> void set(T t) {/*...*/}   // (1a) Error: same erasure
  @Override public <S> S get()  {return null;}   // (2a) Error: same erasure
}
```

```
interface TaskA { Object compute(Integer iRef1); }   // (1)
interface TaskB { Object compute(Integer iRef2); }   // (2)
interface TaskAB extends TaskA, TaskB {              // (3)
  @Override Object compute(Integer iRef3);           // (4) Can be omitted.
}

interface TaskC { String compute(Integer iRef4); }   // (5)
interface TaskABC extends TaskA, TaskB, TaskC {      // (6)
  @Override String compute(Integer iRef5);           // (7a) Can be omitted.
//@Override Object compute(Integer iRef6);           // (7b) Compile-time error!
}
```

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
interface ContractZ extends ContractX, ContractY {
  @Override void doIt(int d);                          // Can be omitted.
  @Override void doIt(double d);                       // Can be omitted.
}
```

```
class Ingredient { }

interface PizzaBakeable {
  void bake(List<Ingredient> ingredients);  // (1) After type erasure: bake(List)
}

interface CalzoneBakeable {
  void bake(List<Ingredient> ingredients);  // (2) After type erasure: bake(List)
}

interface SimplyBakeable {
  void bake(List ingredients);              // (3) Signature: bake(List)
}
```

```
interface Bakeable extends SimplyBakeable, PizzaBakeable, CalzoneBakeable {
  @Override void bake(List ingredients);    // Can be omitted.
}
```

```
Object obj;
...
boolean isIntStack = obj instanceof MyStack<Integer>; // (1) Compile-time error!
boolean isIntStack = obj instanceof MyStack;           // (1')
```

```
IStack<Integer> iStack;
...
boolean isIntegerStack = iStack instanceof MyStack<Integer>;   // (2) OK.
boolean isIntegerStack = iStack instanceof MyStack;   // (2')
boolean isT      = obj instanceof T;                  // (3) Compile-time error!
boolean isTStack = obj instanceof MyStack<T>;         // (4) Compile-time error!
boolean isT      = obj instanceof Object;              // (3')
boolean isTStack = obj instanceof MyStack;             // (4')
boolean isRawStack = obj instanceof MyStack;
boolean isAnyStack = obj instanceof MyStack<?>;       // Preferable.
```

```
Node<Number> numNode = new Node<>(20, null);         // (1)
Node<?> anyNode = numNode;                           // (2)
Node<String> strNode = (Node<String>) anyNode;       // (3) Unchecked cast warning
strNode.setData("Peekaboo");                         // (4)
Number num = numNode.getData();                      // (5) ClassCastException
```

```
Node strNode = (Node) anyNode;                          // (3')
```

```
strNode = (Node<String>) numNode;                   // Compile-time error
```

```
// BiNode<E> is a subtype of MonoNode<E>.
MonoNode<String> monoStrNode = new BiNode<>("Hi", null, null);
BiNode<String> biStrNode = (BiNode<String>) monoStrNode; // Ok. No warning.
```

```
//@SuppressWarnings("unchecked")         // (1) Suppress warnings at (4),(6),(7).
public static void castaway() {
  Object obj = new Node<>("one", null);               // (2)
  Node<String> node1 = obj;                           // (3) Compile-time error!
  Node<String> node2 = (Node<String>) obj;            // (4) Unchecked cast
  Node<String> node3 = (Node<?>) obj;                 // (5) Compile-time error!
  Node<String> node4 = (Node<String>)(Node<?>) obj;  // (6) Unchecked cast
  Node<String> node5 = (Node) obj;                    // (7) Unchecked conversion
  Node<?> node6 = (Node) obj;                         // (8) OK.
  Node<?> node7 = (Node<?>)obj;                       // (9) OK.
}
```

```
String[] strArray = new String[] {"Hi", "Hello", "Howdy"};
Object[] objArray = strArray; // String[] is a subtype of Object[]
objArray[0] = 2020.5;         // (1) ArrayStoreException
```

```
// T is a formal type parameter.
T t = new T();                    // Compile-time error!
T[] anArray = new T[10];          // Compile-time error!
```

```
// An array of Lists of String
List<String>[] list1 = {                          // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<String>[] list2 = new List<String>[] {       // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};
```

```
// An array of Lists of any subtype of Number
List<? extends Number>[] list3
               = new List<? extends Number>[] {    // Compile-time error
  Arrays.asList(20.20, 60.60), Arrays.asList(1948, 1949)
};
```

```
List<?>[] list4 = {
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<?>[] list5 = new List<?>[] {
  Arrays.asList(20.20, 60.60), Arrays.asList(1978, 1981)
};
List[] list6 = list5;
```

```
class MyIntList extends ArrayList<Integer> {}     // A reifiable subclass.

// Client code
List<Integer>[] arrayOfLists = new MyIntList[5];  // Array of Lists of Integer
List<Integer[]> listOfArrays = new ArrayList<>(); // List of Arrays of Integer
```

```
// Copy to array as many elements as possible.
public E[] toArray(E[] toArray) {                                    // (11)
  Node<E> thisNode = tos;
  for (int i = 0; thisNode != null && i < toArray.length; i++) {
    toArray[i] = thisNode.getData();
    thisNode = thisNode.getNext();
  }
  return toArray;
}
```

```
public E[] toArray2() {
  E[] toArray = new E[numOfElements];             // Compile-time error
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

```
public E[] toArray3() {
  E[] toArray = (E[])new Object[numOfElements];  // (1) Unchecked cast warning
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
Integer[] intArray = intStack.toArray3();        // (2) ClassCastException
```

```
@SuppressWarnings("unchecked")
public E[] toArray4(E[] toArray) {
  if (toArray.length != numOfElements) {
    toArray = (E[])java.lang.reflect.Array.newInstance(             // (3)
                         toArray.getClass().getComponentType(),
                         numOfElements);          // Suppressed unchecked warning
  }
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
Integer[] intArray = intStack.toArray4(new Integer[0]);              // OK.
```

```
List<Double>[] arrayOfListsOfDouble
               = (List<Double>[]) new List[1];    // (1) Unchecked cast warning!
arrayOfListsOfDouble[0] = Arrays.asList(10.10);   // (2) Initialize
List<? extends Number>[] arrayOfListsOfExtNums = arrayOfListsOfDouble; // (3)
arrayOfListsOfExtNums[0] = Arrays.asList(10);     // (4) Array storage check ok
Double firstOne = arrayOfListsOfDouble[0].get(0); // (5) ClassCastException!
```

```
public static <T> void
  asStack(MyStack<T> stack, T...elements) { // (1) Possible heap pollution warning
  for (T element : elements) { stack.push(element); }
}
```

```
/* Client code */
// (2) Create a stack of stacks of Integer:
MyStack<MyStack<Integer>> stackOfIntStacks = new MyStack<>();

// (3) Create a stack of Integer:
MyStack<Integer> intStack = new MyStack<>();
intStack.push(2019); intStack.push(2020);

// Initializes the stack of stacks with the stack of Integer.
MyStack.asStack(stackOfIntStacks, intStack); // (4) Unchecked array creation!
intStack = stackOfIntStacks.pop();      // (5) Pop the stack of stacks of Integer.
int tos = intStack.pop();               // (6) Pop the stack of Integer.
assert tos == 2020;
```

```
public static <T> void asStackMalicious(
        MyStack<T> stack, T... elements) { // (1) Possible heap pollution warning
  // Compromise the elements array:
  MyStack<Double> doubleStack = new MyStack<>();
  doubleStack.push(20.20);
  elements[0] = (T) doubleStack;           // (1a) Unchecked cast warning!

  // Copy from elements array:
  for (T element : elements) { stack.push(element); }
}
```

```
public static void asStackMalicious(MyStack stack, Object...elements) {
  // Compromise the elements array:
  MyStack doubleStack = new MyStack();
  doubleStack.push(Double.valueOf(20.20));
  elements[0] = (Object) doubleStack;                        // (1b)
  for (Object element : elements) { stack.push(element); } // (1c)
...
}
```

```
MyStack.asStackMalicious(stackOfIntStacks, intStack); // (4') Unchecked warning!
```

```
asStackMalicious(MyStack, MyStack[])
```

```
@SuppressWarnings("unchecked")                         // (1)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }

@SafeVarargs                                           // (2)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }
```

```
// File: ExceptionErrors.java

// (1) A generic class cannot extend Exception:
class MyGenericException<T> extends Exception { }        // Compile-time error!

public class ExceptionErrors {
  // (2) Cannot specify parameterized types in throws clause:
  public static void main(String[] args)
                   throws MyGenericException<String> { // Compile-time error!
    try {
      throw new MyGenericException<String>();
    } // (3) Cannot use parameterized type in catch block:
      catch (MyGenericException<String> e) {             // Compile-time error!
      e.printStackTrace();
    }
  }
}
```

```java
public interface MyActionListener<E extends Exception> {
  public void doAction() throws E;       // Type parameter in throws clause
}
```

```java
import java.io.FileNotFoundException;

public class FileAction implements MyActionListener<FileNotFoundException> {

  @Override public void doAction() throws FileNotFoundException {
     throw new FileNotFoundException("Does not exist");
  }

  public static void main(String[] args) {
    FileAction fileAction = new FileAction();
    try {
      fileAction.doAction();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

```
class GenericTLC<A> {             // (1) Top-level class

  static class SMC<B> {/*...*/}   // (2) Static member class

  interface SMI<C> {/*...*/}      // (3) Static member interface

  class NSMC<D> {/*...*/}         // (4) Non-static member (inner) class

  void nsm() {
    class NSLC<E> {/*...*/}       // (5) Local (inner) class in non-static context
  }

  static void sm() {
    class SLC<F> {/*...*/}        // (6) Local (inner) class in static context
  }

  // Anonymous classes as parameterized types:
  SMC<A> xsf = new SMC<A>() {     // (7) In non-static context.
    /*...*/                       //     A is type parameter in top-level class.
  };
  SMC<Integer> nsf = new SMC<Integer>() {           // (8) In non-static context
    /*...*/
  };
  static SMI<String> sf = new SMI<String>() {       // (9) In static context
    /*...*/
  };
  SMC<String> nsf1 = new SMC<>() {                  // (10) Using diamond operator
    /*...*/
  };
}
```

```
public class ListPool<T> {       // (1) Top-level class

  static class MyLinkedList<T> { // (2) Hiding type parameter in enclosing context
    T t;                         // T refers to (2)
  }

  class Node<E> {                // (4) Non-static member (inner) class
    T t;                         // T refers to (1)
    E e;
  }

  public static void main(String[] args) {
    // (5) Instantiating a generic top-level class:
    ListPool<String> lp = new ListPool<>();

    // (6) Instantiating a generic static member class:
    ListPool.MyLinkedList<String> list = new ListPool.MyLinkedList<>();

    // (7) Instantiating a generic non-static member class:
    ListPool<String>.Node<Integer> node1 = lp.new Node<>();
    ListPool<String>.Node<Double> node2 = lp.new Node<>();
    ListPool<Integer>.Node<String> node3
                     = new ListPool<Integer>().new Node<>();
  }
}
```

```
enum CoinToss<C> { HEAD, TAIL; }                    // Compile-time error!
enum TripleJump implements java.util.Comparator<TripleJump> {
  HOP, STEP, JUMP;

  @Override public int compare(TripleJump a1, TripleJump a2) {
    return a1.compareTo(a2);
  }
}
```

```
Class<Node> class0 = Node<Integer>.class;// Non-reifiable type. Compile-time error!
Class<Node> class1 = Node.class;        // Reifiable type. Ok.
```

```
Node<Integer> intNode = new Node<>(2019, null);
Node<String> strNode = new Node<>("Hi", null);
Class<?> class2 = strNode.getClass();
Class<?> class3 = intNode.getClass();
assert class1 == class2;
assert class2 == class3;
```

```
import java.util.*;

public class RQ100_00 {
  public static void main(String[] args) {
    List<String> lst1 = new ArrayList<>();
    List<Integer> lst2 = new ArrayList<>();
    List<List<Integer>> lst3 = new ArrayList<>();
    System.out.print(lst1.getClass() + ", ");
    System.out.print(lst2.getClass() + ", ");
    System.out.println(lst3.getClass());
  }
}
```

```
class java.util.ArrayList<String>, class java.util.ArrayList<Integer>, class
java.util.ArrayList<List<Integer>>
```

```
class java.util.ArrayList, class java.util.ArrayList, class
java.util.ArrayList
```

```
class java.util.List, class java.util.List, class java.util.List
class java.util.List<String>, class java.util.List<Integer>, class
java.util.List<List<Integer>>
```

```
import java.util.*;

public class RQ100_02 {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList("Java", "only", "promotes", "fun");
    Collection<String> resultList = delete4LetterWords(lst);
  }

  // (1) INSERT METHOD HEADER HERE
  {
    Collection<E> permittedWords = new ArrayList<>();
    for (E word : words) {
      if (word.length() != 4) permittedWords.add(word);
    }
    return permittedWords;
  }
}
```

```
static <E extends CharSequence>
Collection<? extends CharSequence> delete4LetterWords(Collection<E> words)
```

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<E> words)
```

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<? extends CharSequence> words)
```

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<? extends CharSequence> words)
```

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

```
static <E super CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

```
import java.util.*;

public class RQ_Overloading {

  static <T> void overloadMe(List<T> s1, List<T> s2) {} // (1)
  // (2) INSERT METHOD DECLARATIONS HERE
}
```

```
static <T> void overloadMe(Collection<T> s1, List<T> s2) {}
static <T> void overloadMe(List<T> s1, List<? extends T> s2) {}
static <T> void overloadMe(List<T> s1, Collection<? super T> s2) {}
static <T> void overloadMe(Collection<T> s1, Collection<? super T> s2) {}
static <T> void overloadMe(Collection<T> s1, List<? super T> s2) {}
static <T> void overloadMe(List<? extends T> s1, List<? super T> s2) {}
```

```
import java.util.*;

public class RQ100_87 {
  public static void main(String[] args) {
    List raw = new ArrayList();
    raw.add("2020");
    raw.add(2021);
    raw.add("2022");
    justDoIt(raw);
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

```
class SupC<T> {
  public void set(T t) {/*...*/}        // (1)
  public T get() {return null;}         // (2)
}

class SubC1<M,N> extends SupC<M> {
  public void set(N n) {/*...*/}        // (3)
  public N get() {return null;}         // (4)
}

class SubC2<M,N extends M> extends SupC<M> {
  public void set(N n) {/*...*/}        // (5)
  public N get() {return null;}         // (6)
}
```

```
import java.util.*;

public class CastAway {
  public static void main(String[] args) {
    Object obj = new ArrayList<Integer>();          // (1)
    List<?>       list1 = (List<?>) obj;            // (2)
    List<?>       list2 = (List) obj;               // (3)
    List          list3 = (List<?>) obj;            // (4)
    List<Integer> list4 = (List) obj;               // (5)
    List<Integer> list5 = (List<Integer>) obj;      // (6)
  }
}
```

```
import java.util.*;

public class InstanceTest2 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    Set<Double> doubleSet = new HashSet<>();
    List<?>          list = intList;
    Set<?>            set = doubleSet;

    scuddle(intList);
    scuddle(doubleSet);
    scuddle(list);
    scuddle(set);
  }
  private static void scuddle(Collection<?> col) {
    if (col instanceof List<?>) {
      System.out.println("I am a list.");
    } else if (col instanceof Set<?>) {
      System.out.println("I am a set.");
    }
  }
}
```

```
public class GenArrays {
  public static <E> E[] copy(E[] srcArray) {
    E[] destArray = (E[]) new Object[srcArray.length];
    int i = 0;
    for (E element : srcArray) {
      destArray[i++] = element;
    }
    return destArray;
  }

  public static void main(String[] args) {
    String[] sa = {"9", "1", "1" };
    String[] da = GenArrays.copy(sa);
    System.out.println(da[0]);
  }
}
```

```
import java.util.*;

public class GenVarArgs {
  public static <T> void doIt(List<T>... aols) {           // (1)
    for (int i = 0; i < aols.length; i++) {
      System.out.print(aols[i] + " ");
    }
  }

  public static void main(String... args) {                // (2)
    List<String> ls1 = Arrays.asList("one", "two");
    List<String> ls2 = Arrays.asList("three", "four");
    List<String>[] aols = new List[] {ls1, ls2};           // (3)
    doIt(aols);                                            // (4)
  }
}
```

```
ArrayList<String> palindromes = new ArrayList<String>(); // (1)
```

```
ArrayList<String> palindromes = new ArrayList<>(); // Using the diamond operator
```

```
ArrayList<Integer> intList = new ArrayList<>();        // (1) ArrayList of Integer
intList.add(10); intList.add(100); intList.add(1000);
```

```
ArrayList<String> newList1 = new ArrayList(intList);   // (2) Unchecked conversion
                                                       //     warning
System.out.println(newList1.get(0));                   // (3) ClassCastException!

ArrayList<String> newList2 = new ArrayList<>(intList); // (4) Compile-time error!
```

```
List<String> palindromes = new ArrayList<>();    // (2) List<String> reference
```

```
List<String> palindromes = new LinkedList<>(); // Changing implementation.
```

```
List<String> palindromes = new ArrayList<>(20); // Initial capacity is 20.
```

```
palindromes.add("level"); palindromes.add("Ada"); palindromes.add("kayak");
System.out.println(palindromes);
```

```
List<String> wordList = new ArrayList<>(palindromes);
System.out.println(wordList); // [level, Ada, kayak]
wordList.add("Naan");
System.out.println(wordList); // [level, Ada, kayak, Naan]
```

```
List<StringBuilder> synonyms   = new ArrayList<>(); // List of StringBuilder
List<Integer> attendance       = new ArrayList<>(); // List of Integer
List<List<String>> listOfLists = new ArrayList<>(); // (3) List of List of String
List<int> frequencies          = new ArrayList<>(); // (4) Compile-time error!
```

```
Object[] objArray = new String[10];                    // (5) OK!
```

```
objArray[2] = "Green";                              // (6) OK!
objArray[1] = Integer.valueOf(2016);                // ArrayStoreException!
```

```
ArrayList<Object> objList1 = new ArrayList<String>();// (7) Compile-time error!
List<Object> objList2 = new ArrayList<String>();     // (8) Compile-time error!
```

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

```
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5,
                      E e6, E e7, E e8, E e9, E e10)
```

```
@SafeVarargs static <E> List<E> of(E... elements)
```

```
static <E> List<E> copyOf(Collection<? extends E> collection)
```

```
List<String> list = List.of("Tom", "Dick", "Harriet");
//  list.add("Harry");                          // UnsupportedOperationException
//  list.remove(2);                             // UnsupportedOperationException
//  list.set(0, "Tommy");                       // UnsupportedOperationException
System.out.println(list);                       // [Tom, Dick, Harriet]
System.out.println(list instanceof ArrayList);  // false
```

```
List<String> coinList = List.of("nickel", "dime", null); // NullPointerException
```

```
List<Integer> intList1 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);   // Fixed-arity
List<Integer> intList2 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Varargs
System.out.println(intList1);        // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
System.out.println(intList2);        // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
String[] strArray = {"Tom", "Dick", "Harriet"};
List<String> strList = List.of(strArray);                    // (1) List of String
List<String[]> strArrayList = List.<String[]>of(strArray); // (2) List of String[]
System.out.println(strList);       // [Tom, Dick, Harriet]
System.out.println(strArrayList);  // [[Ljava.lang.String;@3b22cdd0]
```

```
List<String> fab4 = new ArrayList<>();
fab4.add("John"); fab4.add("Paul"); fab4.add("George"); fab4.add("Ringo");
System.out.println(fab4);                     // [John, Paul, George, Ringo]
List<String> fabAlways = List.copyOf(fab4);   // (1)
fab4.remove("John"); fab4.remove("George");   // Modify original list
System.out.println(fab4);                     // [Paul, Ringo]
System.out.println(fabAlways);                // [John, Paul, George, Ringo]
```

`boolean add(E element)`	From `List<E>` interface.
`void add(int index, E element)`	From `List<E>` interface.

`boolean addAll(Collection<? extends E> c)`	From `List<E>` interface.
`boolean addAll(int index, Collection<? extends E> c)`	From `List<E>` interface.

`E set(int index, E element)`	From `List<E>` interface.

`default void replaceAll(UnaryOperator<E> operator)`	From `List<E>` interface.

`void clear()`	From `List<E>` interface.

`E remove(int index)`	From `List<E>` interface.
`boolean remove(Object element)`	From `List<E>` interface.

`boolean removeAll(Collection<?> c)`	From `List<E>` interface.
`boolean removeIf(Predicate<? super E> filter)`	From `Collection<E>` interface.

`void ensureCapacity(int minCapacity)`	From `List<E>` interface.

```
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

```
System.out.println("\n(2) Add elements to list:");
for (String str : wordArray) {
  strList.add(str);
  printListWithIndex(strList);
}
```

```
                              // [0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
strList.add(2, "Java");       // Insert an element at index 2 in the list.
printListWithIndex(strList); // [0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob,
                              //  5:Rotator, 6:Bob]
```

```
List<String> list1 = new ArrayList<>();   // []
list1.add(0, "First");                    // [First]
list1.add(list1.size(), "Last");          // [First, Last]
```

```
System.out.println("(3) Replace the element at index 1:");
String oldElement = strList.set(1, "Naan");
System.out.println("Element that was replaced: " + oldElement);   // "Ada"
printListWithIndex(strList); // [0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob,
                             //  5:Rotator, 6:Bob]
// list1 is [First, Last].
list1.clear();                                    // []
```

```
System.out.println("(4) Remove the element at index 0:");
System.out.println("Element removed: " + strList.remove(0));      // "level"
printListWithIndex(strList); // [0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

System.out.println("(5) Remove the first occurrence of \"Java\":");
System.out.println("Element removed: " + strList.remove("Java")); // true
printListWithIndex(strList); // [0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]
```

```
List<StringBuilder> sbList = new ArrayList<>();
for (String str : wordArray)
  strList.add(str);
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
StringBuilder element = new StringBuilder("Ada");
System.out.println("Element to be removed: " + element);          // "Ada"
System.out.println("Element removed: " + sbList.remove(element)); // false
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
```

```
sbList.trimToSize();           // Capacity is now same as size—that is, 6.
sbList.ensureCapacity(10000);  // Capacity is now large enough for 10000 elements.
                               // Size is still 6.
```

```
List<Integer> intList = new ArrayList<>();
intList.add(10); intList.add(20); intList.add(1);
System.out.println(intList);                                // [10, 20, 1]
System.out.println("Element to be removed: " + 1);          // 1
System.out.println("Element removed: " + intList.remove(1));  // 20
System.out.println(intList);                                  // [10, 1]
System.out.println(intList);                                 // [10, 20, 1]
System.out.println("Element to be removed: " + 1);                     // 1
System.out.println("Element removed: " +
                   intList.remove(Integer.valueOf(1)));      // true
System.out.println(intList);                                 // [10, 20]
```

```
intList.remove(Integer.valueOf(1))
```

`int size()`	From `List<E>` interface.

`boolean isEmpty()`	From `List<E>` interface.

`E get(int index)`	From `List<E>` interface.

`boolean contains(Object element)`	From `List<E>` interface.

`int indexOf(Object o)`	From `List<E>` interface.
`int lastIndexOf(Object o)`	From `List<E>` interface.

`List<E> subList(int fromIndex, int toIndex)`	From `List<E>` interface.

`boolean equals(Object o)`	From `List<E>` interface.

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("The size of the list is " + strList.size());          // 5
boolean result = strList.isEmpty();
System.out.println("The list " + (result ? "is" : "is not") + " empty."); // false
```

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("First element: " + strList.get(0);                 // Naan
System.out.println("Last element: " + strList.get(strList.size()-1));  // Bob
```

```
List<String> strList2 = new ArrayList<>(strList);
boolean trueOrFalse = strList.equals(strList2);                      // true
```

```
out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator, Bob]
List<String> strList3 = strList.subList(1, 4);
out.println("Sublist before remove: " + strList3);    // [kayak, Bob, Rotator]
out.println("Remove: " + strList3.get(0));  // "kayak"
strList3.remove(0);                         // Remove element at index 0
out.println("Sublist after remove: " + strList3);     // [Bob, Rotator]
out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]
```

```
boolean found = strList.contains("Naan");     // true
int index = strList.indexOf("Bob");           // 2
index = strList.indexOf("BOB");               // -1 (Not found)
index = strList.lastIndexOf("Bob");           // 4 (Last occurrence)
```

```
public static <E> void printListWithIndex(List<E> list) {
  List<String> newList = new ArrayList<>();
  for (int i = 0; i < list.size(); i++) {
    newList.add(i + ":" + list.get(i));
  }
  System.out.println(newList);
}
```

```
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
```

```
for (String str : strList) {
  if (str.length() <= 3) {
    strList.remove(str);                // Throws ConcurrentModificationException
  }
}
```

`Object[] toArray()`	From `Collection<E>` interface.
`<T> T[] toArray(T[] a)`	From `Collection<E>` interface.

`default <T> T[]` `toArray(IntFunction<T[]> generator)`	From `Collection<E>` interface.

```
System.out.println("(15) Convert list to array:");
Object[] objArray = strList.toArray();                      // Object[]
System.out.println("Object[] length: " + objArray.length); // 5
System.out.print("Length of each string in the Object array: ");
for (Object obj : objArray) {
  String str = (String) obj;                                // Cast required.
  System.out.print(str.length() + " ");
}
System.out.println();
```

```
String[] strArray = strList.toArray(new String[0]);         // String[]
System.out.println("String[] length: " + strArray.length);  // 5
System.out.print("Length of each string in the String array: ");
for (String str : strArray) {
  System.out.print(str.length() + " ");
}
System.out.println();
```

`@SafeVarargs <E> List<E> asList(E... elements)`	From `Arrays` class.

```
List<Integer> list2 = Arrays.asList(9, 1, 1);          // (2) Varargs

int[] array2 = new int[] {9, 1, 1};                    // An array of int
// List<Integer> intList3 = Arrays.asList(array2);     // (3) Compile-time error!
```

```
System.out.println(list1);                     // [9, 1, 1]
// list1.add(10);                              // (4) UnsupportedOperationException
list1.set(0, 10);                              // (5)
System.out.println(list1);                     // (6) [10, 1, 1]
System.out.println(Arrays.toString(array1)); // (7) [10, 1, 1]

List<Integer> sublist1 = list1.subList(0, 2);// (8)
System.out.println(sublist1);                  // [10, 1]
// sublist1.clear();                           // (9) UnsupportedOperationException
Collections.sort(sublist1);                    // (10)
System.out.println(sublist1);                  // [1, 10]
System.out.println(list1);                     // [1, 10, 1]
System.out.println(Arrays.toString(array1)); // [1, 10, 1]
```

```
String[] jiveArray   = new String[] {"java", "jive", "java", "jive"};
Set<String> jiveSet  = new HashSet<>(Arrays.asList(jiveArray));// (1)
String[] uniqueJiveArray = jiveSet.toArray(new String[0]);     // (2)
System.out.println(Arrays.toString(uniqueJiveArray));          // (3) [java, jive]
```

```
Integer[] yrArray1 = {2020, 2021, 2022};
List<Integer> yrlist1 = Arrays.asList(yrArray1);
yrArray1[0] = 2019;                                  // Modify the array
out.println("yrArray1: " + Arrays.toString(yrArray1)); //     [2019, 2021, 2022]
out.println("yrlist1: " + yrlist1);                  // (1) [2019, 2021, 2022]

Integer[] yrArray2 = {2020, 2021, 2022};
List<Integer> yrlist2 = List.of(yrArray2);
yrArray2[0] = 2019;                                  // Modify the array
out.println("yrArray2: " + Arrays.toString(yrArray2)); //     [2019, 2021, 2022]
out.println("yrlist2: " + yrlist2);                  // (2) [2020, 2021, 2022]
```

```
List<Integer> yrList3 = Arrays.asList(2020, 2021, 2022);
yrList3.set(2, 2023);                    // (1) OK
out.println(yrList3);                    // [2020, 2021, 2023]

List<Integer> yrlist4 = List.of(2020, 2021, 2022);
yrlist4.set(2, 2023);                    // (2) UnsupportedOperationException
```

```
yrList3.add(2050);                                  // UnsupportedOperationException
yrlist4.remove(0);                                  // UnsupportedOperationException
```

```
List<Integer> yrList5 = Arrays.asList(2020, 2021, null);  // OK.
List<Integer> yrlist6 = List.of(2020, 2021, null);        // NullPointerException
```

```
boolean flag1 = Arrays.asList(2021, 2022).contains(null); // OK.
boolean flag2 = List.of(2021, 2022).contains(null);       // NullPointerException
```

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class ArrayListMethods {

  public static void main(String[] args) {

    String[] wordArray = { "level", "Ada", "kayak", "Bob", "Rotator", "Bob" };

    out.println("(1) Create an empty list of strings:");
    List<String> strList = new ArrayList<>();
    printListWithIndex(strList);

    out.println("\n(2) Add elements to list:");
    for (String str : wordArray) {
      strList.add(str);
      printListWithIndex(strList);
    }
    out.println("Insert an element at index 2 in the list:");
    strList.add(2, "Java");
    printListWithIndex(strList);

    out.println("\n(3) Replace the element at index 1:");
    String oldElement = strList.set(1, "Naan");
    out.println("Element that was replaced: " + oldElement);
    printListWithIndex(strList);
```

```
out.println("\n(4) Remove the element at index 0:");
out.println("Element removed: " + strList.remove(0));
printListWithIndex(strList);

out.println("\n(5) Remove the first occurrence of \"Java\":");
out.println("Element removed: " + strList.remove("Java"));
printListWithIndex(strList);

out.println("\n(6) Determine the size of the list:");
out.println("The size of the list is " + strList.size());

out.println("\n(7) Determine if the list is empty:");
boolean result = strList.isEmpty();
out.println("The list " + (result ? "is" : "is not") + " empty.");

out.println("\n(8) Get the element at specific index:");
out.println("First element: " + strList.get(0));
out.println("Last element: " + strList.get(strList.size() - 1));

out.println("\n(9) Compare two lists:");
List<String> strList2 = new ArrayList<>(strList);
boolean trueOrFalse = strList.equals(strList2);
out.println("The lists strList and strList2 are"
    + (trueOrFalse ? "" : " not") + " equal.");
strList2.add(null);
printListWithIndex(strList2);
trueOrFalse = strList.equals(strList2);
out.println("The lists strList and strList2 are"
    + (trueOrFalse ? "" : " not") + " equal.");

out.println("\n(10) Sublists as views:");
out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator, Bob]
List<String> strList3 = strList.subList(1, 4);
out.println("Sublist before remove: " + strList3);    // [kayak, Bob, Rotator]
out.println("Remove: " + strList3.get(0));  // "kayak"
strList3.remove(0);                         // Remove element at index 0
out.println("Sublist after remove: " + strList3);     // [Bob, Rotator]
out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]

out.println("\n(11) Membership test:");
boolean found = strList.contains("Naan");
String msg = found ? "contains" : "does not contain";
out.println("The list " + msg + " the string \"Naan\".");

out.println("\n(12) Find the index of an element:");
int pos = strList.indexOf("Bob");
out.println("The index of string \"Bob\" is: " + pos);
pos = strList.indexOf("BOB");
out.println("The index of string \"BOB\" is: " + pos);
pos = strList.lastIndexOf("Bob");
out.println("The last index of string \"Bob\" is: " + pos);
printListWithIndex(strList);

out.println("\n(13) Iterating over the list using the for(;;) loop:");
for (int i = 0; i < strList.size(); i++) {
  out.print(i + ":" + strList.get(i) + " ");
```

```
    }
    out.println();

    out.println("\n(14) Iterating over the list using the for(:) loop:");
    for (String str : strList) {
      out.print(str +  " ");
      // strList.remove(str);        // Throws ConcurrentModificationException.
    }
    out.println();

    out.println("\n(15) Convert list to array:");
    Object[] objArray = strList.toArray();
    out.println("Object[] length: " + objArray.length);
    out.print("Length of each string in the Object array: ");
    for (Object obj : objArray) {
      String str = (String) obj; // Cast required.
      out.print(str.length() + " ");
    }
    out.println();
    String[] strArray = strList.toArray(new String[0]);
    out.println("String[] length: " + strArray.length);
    out.print("Length of each string in the String array: ");
    for (String str : strArray) {
      out.print(str.length() + " ");
    }
  }

  /**
   * Print the elements of a list, together with their index:
   * [0:value0, 1:value1, ...]
   * @param list   List to print with index
   */
  public static <E> void printListWithIndex(List<E> list) {            // (16)
    List<String> newList = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
      newList.add(i + ":" + list.get(i));
    }
    out.println(newList);
  }
}
```

```
(1) Create an empty list of strings:
[]

(2) Add elements to list:
[0:level]
[0:level, 1:Ada]
[0:level, 1:Ada, 2:kayak]
[0:level, 1:Ada, 2:kayak, 3:Bob]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
Insert an element at index 2 in the list:
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

```
(3) Replace the element at index 1:
Element that was replaced: Ada
[0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]

(4) Remove the element at index 0:
Element removed: level
[0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

(5) Remove the first occurrence of "Java":
Element removed: true
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]

(6) Determine the size of the list:
The size of the list is 5

(7) Determine if the list is empty:
The list is not empty.

(8) Get the element at specific index:
First element: Naan
Last element: Bob

(9) Compare two lists:
The lists strList and strList2 are equal.
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob, 5:null]
The lists strList and strList2 are not equal.

(10) Sublists as views:
Underlying list: [Naan, kayak, Bob, Rotator, Bob]
Sublist before remove: [kayak, Bob, Rotator]
Remove: kayak
Sublist after remove: [Bob, Rotator]
Underlying list: [Naan, Bob, Rotator, Bob]

(11) Membership test:
The list contains the string "Naan".

(12) Find the index of an element:
The index of string "Bob" is: 1
The index of string "BOB" is: -1
The last index of string "Bob" is: 3
[0:Naan, 1:Bob, 2:Rotator, 3:Bob]

(13) Iterating over the list using the for(;;) loop:
0:Naan 1:Bob 2:Rotator 3:Bob

(14) Iterating over the list using the for(:) loop:
Naan Bob Rotator Bob

(15) Convert list to array:
Object[] length: 4
Length of each string in the Object array: 4 3 7 3
String[] length: 4
Length of each string in the String array: 4 3 7 3
```

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A10 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    for (int i = 0; i < strList.size(); /* empty */) {
      if (strList.get(i).length() <= 3) {
        strList.remove(i);
      } else {
        ++i;
      }
    }
    System.out.println(strList);
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A15 {
  public static void main(String[] args) {
    doIt1(); doIt2();
  }

  public static void doIt1() {
    List<StringBuilder> sbListOne = new ArrayList<>();
    sbListOne.add(new StringBuilder("Anna"));
    sbListOne.add(new StringBuilder("Ada"));
    sbListOne.add(new StringBuilder("Bob"));
    List<StringBuilder> sbListTwo = new ArrayList<>(sbListOne);
    sbListOne.add(null);
    sbListTwo.get(1).reverse();
    System.out.println(sbListOne);                                  // (1)
  }

  public static void doIt2() {
    List<String> listOne = new ArrayList<>();
    listOne.add("Anna"); listOne.add("Ada"); listOne.add("Bob");
    List<String> listTwo = new ArrayList<>(listOne);
    String strTemp = listOne.get(0);
    listOne.set(0, listOne.get(listOne.size()-1));
    listOne.set(listOne.size()-1, strTemp);
    System.out.println(listTwo);                                     // (2)
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A20 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        System.out.print(i);
      }
    }
    System.out.println();
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A30 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        strList.remove(i);
      }
    }
    System.out.println(strList);
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class RQ12A40 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    while (strList.remove("Bob"));
    System.out.println(strList);
  }
}
```

```
import java.util.*;
public class Test12_01 {
  public static void main(String[] args) {
    String[] data1 = {"A","B","B","A"};
    List<String> data2 = new ArrayList<>();
    for (String s : data1) {
      data2.add(s);
    }
    data2.set(1, "X");
    data2.add(1, "X");
    data2.remove(2);
    System.out.println(data2);
  }
}
```

```
import java.util.*;
public class Test12_02 {
  public static void main(String[] args) {
    String[] data1 = {"A","B","B","A"};
    List<String> data2 = Arrays.asList(data1);
    data2.set(1, "X");
    data2.set(2, "X");
    System.out.println(data2);
  }
}
```

```
public class Song {
  private String name;
  public Song(String name) {
    this.name = name;
  }
  public void update() {
    name = name.toUpperCase();
  }
  public String toString() {
    return name;
  }
}
```

```
import java.util.*;
public class Test12_04 {
  public static void main(String[] args) {
    Song[] playArray1 = {new Song("a"), new Song("b")};
    List<Song> playlist = Arrays.asList(playArray1);
    Song[] playArray2 = playlist.toArray(new Song[]{});
    playArray1[1].update();
    System.out.print(playArray1[1]);
```

```
      System.out.print(playlist.get(1));
      System.out.print(playArray2[1]);
    }
  }
```

```
public class MySong {
  private String name;
  public MySong(String name) {
    this.name = name;
  }
  public String toString() {
    return name;
  }
}
```

```
import java.util.*;
public class Test12_05 {
  public static void main(String[] args) {
    MySong[] playArray1 = {new MySong("A"), new MySong("B")};
    List<MySong> playlist = List.of(playArray1);
    MySong[] playlist2 = playlist.toArray(new MySong[]{});
    playArray1[0] = new MySong("C");
    System.out.print(playArray1[0]);
    System.out.print(playlist.get(0));
    System.out.print(playlist2[0]);
  }
}
```

```
import java.util.*;
public class Test12_06 {
  public static void main(String[] args) {
    List<String> data1 = List.of("A","B","C");
    String[] data2 = data1.toArray(new String[]{"X","Y","Z"});
    data2[1] = data1.get(0).toLowerCase();
    for (String s: data2) {
      System.out.print(s);
    }
  }
}
```

```
import java.util.*;
public class Test12_07 {
  public static void main(String[] args) {
    List<Character> text = new ArrayList<>(3);
    for (char a = 'a'; a <= 'e'; a++) {
      text.add(a);
    }
    System.out.println(text);
  }
}
```

```
@FunctionalInterface                                  // Annotation.
interface StrPredicate {
  boolean test(String str);                           // Sole public abstract method.
}
```

```
@FunctionalInterface
interface CharSeqPredicate {                                     // Compile-time error!
  boolean test(String str);                                      // Abstract method.
  boolean test(StringBuilder sb);                                // Abstract method.
}
```

```
@FunctionalInterface
interface NewStrPredicate {
  boolean test(String str);                                 // (1) Abstract method
  default void msg(String str) { System.out.println(str); } // (2) Default method
  static void info() { System.out.println("Testing!"); }    // (3) Static method
}
```

```
@FunctionalInterface
interface Funky {
  @Override int hashCode();                              // From Object class
  @Override boolean equals(Object obj);                  // From Object class
  @Override String toString();                           // From Object class
  boolean doTheFunk(Object obj);                         // Abstract method
}
```

```
@FunctionalInterface
interface Predicate<T> {
  boolean test(T element);                                 // Functional method.
  // ...
}
```

```
public static <T> boolean testPredicate(T object, Predicate<T> predicate) {
  return predicate.test(object);
}
```

```
// Class implementation of Predicate<String>.
class PredicateTest implements Predicate<String> {
  public boolean test(String str) {
    return str.startsWith("A");              // (1)
  }
}

// An anonymous class implementation of Predicate<String>.
static Predicate<String> testLength = new Predicate<>() {
  public boolean test(String str) {
    return str.length() < 4;                 // (2)
  }
};

// Client code:
System.out.println(testPredicate("Anna", new PredicateTest()));   // true
System.out.println(testPredicate("Anna", testLength));            // false
```

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
interface ContractZ extends ContractX, ContractY {
  @Override void doIt(int d);                              // Can be omitted.
  @Override void doIt(double d);                           // Can be omitted.
}
```

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
@FunctionalInterface interface ContractZ                   // Compile-time error!
                     extends ContractX, ContractY {
  @Override void doIt(int d);                              // Compile-time error!
  @Override void doIt(double d);                           // Compile-time error!
}
```

```
interface Feature1<R> { void flatten(List<R> plist); }
interface Feature2<S> { void flatten(List<S> plist); }
@FunctionalInterface
interface Features<T> extends Feature1<T>, Feature2<T> {   // (1)
  @Override void flatten(List<T> plist);                   // Can be omitted.
}
```

```
(Integer a, String y) -> {};            // Multiple declared-type parameters
```

```
(a, b) -> {};                             // Multiple inferred-type parameters
```

```
(var a, var b) -> {};                   // Multiple var-type inferred parameters
```

```
() -> {};                         // Empty parameter list

// Single formal parameter:
(String str) -> {};               // Single declared-type parameter
(str)        -> {};               // Single inferred-type parameter
str          -> {};               // Single inferred-type parameter
(var str)    -> {};               // Single var-type inferred parameter

// Multiple formal parameters:
(Integer x, Integer y) -> {};     // Multiple declared-type parameters
(x, y)                 -> {};     // Multiple inferred-type parameters
(var x, var y)         -> {};     // Multiple var-type inferred parameters

// Modifiers and annotations with formal parameters:
(final int i, int j) -> {};       // Modifier with declared-type parameters
(final var i, var j) -> {};       // Modifier with var-type inferred parameters
(@NonNull int i, int j) -> {};    // Annotation with declared-type parameter
(var i, @Nullable var j)-> {};    // Annotation with var-type inferred parameter

// Parentheses are mandatory with multiple formal parameters:
String str           -> {};       // Illegal: Missing parentheses
var str              -> {};       // Illegal: Missing parentheses
Integer x, Integer y -> {};       // Illegal: Missing parentheses
x, y                 -> {};       // Illegal: Missing parentheses
var x, var y         -> {};       // Illegal: Missing parentheses

// All formal parameters must be either declared-type, inferred-type, or
// var-type inferred parameters.
(String str, j)     -> {};      // Cannot mix declared-type and inferred-type
(String str, var j) -> {};      // Cannot mix declared-type and var-type inferred
(var str, j)        -> {};      // Cannot mix var-type inferred and inferred-type

// Modifiers and annotations cannot be used with inferred-type parameters.
(final str, j)      -> {};      // No modifiers with inferred-type parameters
(str, @NonNull j)   -> {};      // No annotations with inferred-type parameters
```

```
() -> 2021                                  // Expression body, non-void return
() -> null                                  // Expression body, non-void return
(i, j) -> i + j                             // Expression body, non-void return
(i, j) -> i <= j ? i : j                    // Expression body, non-void return
str -> str.length() > 3                     // Expression body, non-void return
str -> str != null                          // Expression body, non-void return
       && !str.equals("") && str.length() > 3
       &&  str.equals(new StringBuilder(str).reverse().toString())
```

```
val -> System.out.println(val)    // Method invocation statement, void return
sb -> sb.trimToSize()             // Method invocation statement, void return
sb -> sb.append("!")              // Method invocation statement, non-void return
() -> new StringBuilder("?")      // Object creation statement, non-void return
value -> value++                  // Increment statement, non-void return
value -> value *= 2               // Assignment statement, non-void return
```

```
(int i) -> while (i < 10) ++i      // Illegal: not an expression but statement
(x, y) -> return x + y             // Illegal: return not allowed in expression
```

```
() -> {}                                    // Block body, void return
() -> { return 2021; }                      // Block body, non-void return
() -> { return 2021 }      // Illegal: statement terminator (;) in block missing
() -> { new StringBuilder("Go nuts."); }           // Block body, void return
() -> { return new StringBuilder("Go nuts!"); }    // Block body, non-void return
(int i) -> { while (i < 10) ++i; }                 // Block body, void return
(i, j) -> { if (i <= j) return i; else return j; } // Block body, non-void return
(done) -> {                     // Multiple statements in block body, void return
  if (done) {
    System.out.println("You deserve a break!");
    return;
  }
  System.out.println("Stay right here!");
}
```

```
Predicate<Integer> p1 = i -> i % 2 == 0;  // (1) Target type: Predicate<Integer>
```

```
public boolean test(Integer t);          // Method type: Integer -> boolean
```

```
Predicate<Integer> p1 = new Predicate<>() {  // Anonymous class
  public boolean test(Integer i) {
    return i % 2 == 0;
  }
};
```

```
IntPredicate p2 = i -> i % 2 == 0;         // (2) Target type: IntPredicate
```

```
public boolean test(int i);              // Method type: int -> boolean
```

```
System.out.println(p1 == p2);              // false
```

```
boolean result1 = p1.test(2021);          // false
boolean result2 = p2.test(2020);          // true
```

```
List: [Tom, Dick, Harriet]
```

```
p.test("never dies!") + " " + obj.banner
```

```
true + " " + "love never dies!"
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class MembersOnly {

  // Instance variable
  private StringBuilder banner;                                     // (1)

  // Static variable
  private static List<String> strList;                              // (2)

  // Constructor
  public MembersOnly(String str) {
    banner = new StringBuilder(str);
  }

  // Static method
  public static void main(String[] args) {
    strList = new ArrayList<>();                                    // (3)
    strList.add("Tom"); strList.add("Dick"); strList.add("Harriet");

    MembersOnly obj = new MembersOnly("love ");                     // (4)
    Predicate<String> p = obj.getPredicate();                       // (5)
    System.out.println(p.test("never dies!") + " " + obj.banner);  // (6)
  }

  // Instance method
  public Predicate<String> getPredicate() {               // (7)
    return str -> {                                       // (8)  Lambda expression
      System.out.println("List: " + MembersOnly.strList);// (9)  Static field
      this.banner.append(str);                            // (10) Instance field
      return str.length() > 5;                            // (11) boolean value
    };
  }
}
```

```
List: [Tom, Dick, Harriet]
true love never dies!
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LocalsOnly {

  public static void main(String[] args) {
    StringBuilder banner = new StringBuilder("love ");
    LocalsOnly instance = new LocalsOnly();
    Predicate<String> p = instance.getPredicate(banner);
    System.out.println(p.test("never dies!") + " " + banner);
  }

  public Predicate<String> getPredicate(StringBuilder banner) {   // (1)
    List<String> words = new ArrayList<>();                       // (2)
    words.add("Tom"); words.add("Dick"); words.add("Harriet");

//  banner = new StringBuilder();         // (3) Illegal: Not effectively final
//  words = new ArrayList<>();            // (4) Illegal: Not effectively final

    return str -> {                                     // (5) Lambda expression
//    String banner = "Don't redeclare me!";            // (6) Illegal: Redeclared
//    String[] words = new String[6];                   // (7) Illegal: Redeclared
      System.out.println("List: " + words);             // (8) Local variable
      banner.append(str);                               // (9) Parameter
      return str.length() > 5;
    };
  }
}
```

```
List: [Tom, Dick, Harriet]
true love never dies!
```

```
str.equals(new StringBuilder(str).reverse().toString())
str.equalsIgnoreCase(new StringBuilder(str).reverse().toString())
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FunWithPalindromes {

  public static void main(String[] args) {

    List<String> words = new ArrayList<>();
    words.add("Otto"); words.add("ADA"); words.add("Alyla");
    words.add("Bob"); words.add("HannaH"); words.add("Java");
    System.out.println("List of words:                " + words);

    System.out.println("-----------Using Anonymous Classes---------------------");
    // Use an anonymous class to filter for palindromes (case sensitive).  (1)
    List<String> palindromesA = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equals(new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-sensitive palindromes:   " + palindromesA);

    // Use an anonymous class to filter for palindromes (case insensitive). (2)
    List<String> palindromesB = filterList(words,
        new Predicate<String>() {
          @Override public boolean test(String str) {
            return str.equalsIgnoreCase(
                           new StringBuilder(str).reverse().toString());
          }
        }
    );
    System.out.println("Case-insensitive palindromes: " + palindromesB);
    System.out.println("-----------Using Lambda Expressions--------------------");
    Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (3)
    List<String> palindromes1 = filterList(words, predicate1);            // (4)
    System.out.println("Case-sensitive palindromes:   " + palindromes1);

    Predicate<String> predicate2 = str ->
        str.equalsIgnoreCase(new StringBuilder(str).reverse().toString());// (5)
    List<String> palindromes2 = filterList(words, predicate2);            // (6)
    System.out.println("Case-insensitive palindromes: " + palindromes2);
  }

  /**
   * Filters a list according to the criteria of the predicate.
   * @param list       List to filter
   * @param predicate  Provides the criteria for filtering the list
   * @return           List of elements that match the criteria
   */
  public static <E> List<E> filterList(List<E> list,                        // (7)
                                       Predicate<E> predicate) {
    List<E> result = new ArrayList<>();
    for (E element : list) {
      if (predicate.test(element)) {                                      // (8)
        result.add(element);
      }
    }
    return result;
  }
}
```

```
List of words:                [Otto, ADA, Alyla, Bob, HannaH, Java]
-----------Using Anonymous Classes---------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
-----------Using Lambda Expressions--------------------
Case-sensitive palindromes:   [ADA, HannaH]
Case-insensitive palindromes: [Otto, ADA, Alyla, Bob, HannaH]
```

```
Predicate<String> predicate1 = str ->
        str.equals(new StringBuilder(str).reverse().toString());          // (3)
```

```
new Predicate<String>() {
  @Override public boolean test(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
  }
}
```

```
str -> str.equals(new StringBuilder(str).reverse().toString())
```

```
interface Funky1    { void    absMethod1(String s); }
interface Funky2    { String  absMethod2(String s); }

public class RQ12A99 {
  public static void main(String[] args) {

    Funky1 p1;
    p1 = s -> System.out.println(s);           // (1)
    p1 = s -> s.length();                      // (2)
    p1 = s -> s.toUpperCase();                      // (3)
    p1 = s -> { s.toUpperCase(); };                 // (4)
    p1 = s -> { return s.toUpperCase(); };          // (5)

    Funky2 p2;
    p2 = s -> System.out.println(s);                // (6)
    p2 = s -> s.length();                           // (7)
    p2 = s -> s.toUpperCase();                      // (8)
    p2 = s -> { s.toUpperCase(); };                 // (9)
    p2 = s -> { return s.toUpperCase(); };          // (10)
  }
}
```

```
interface AgreementA { void doIt(); }
interface AgreementB extends AgreementA {}
interface AgreementC extends AgreementB {
  void doIt();
  boolean equals(Object obj);
}

class Beta implements AgreementB {
  public void doIt() {
    System.out.print("Jazz|");
  }
}

public class RQ12A999 {
  public static void main(String[] args) {
    AgreementA a = () -> System.out.print("Java|");        // (1)
    AgreementB b = () -> System.out.print("Jive|");        // (2)
    AgreementC c = () -> System.out.print("Jingle|");      // (3)
    Object o = a = c;                                      // (4)
    b = new Beta();                                        // (5)
    a.doIt();                                              // (6)
    b.doIt();                                              // (7)
    c.doIt();                                              // (8)
    ((AgreementA) o).doIt();                               // (9)
  }
}
```

```
Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   // (1)
System.out.println(createSB.get());                              // Prints: Howdy!

String str = "uppercase me!";
Supplier<String> makeUppercase = () -> str.toUpperCase();              // (2)
System.out.println(makeUppercase.get());              // Prints: UPPERCASE ME!
```

```
Random numGen = new Random();

Supplier<Integer> intSupplier = () -> numGen.nextInt(100); // numGen effect. final
System.out.println(intSupplier.get());             // Prints a number in [0, 100).
```

```
List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
```

```
IntSupplier intSupplier2 = () -> numGen.nextInt(100);
System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 100).
```

```
roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
```

```
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.function.DoubleSupplier;
import java.util.function.IntSupplier;
import java.util.function.Supplier;

public class SupplierClient {
  public static void main(String[] args) {

    Supplier<StringBuilder> createSB = () -> new StringBuilder("Howdy!");   // (1)
    System.out.println(createSB.get());                            // Prints: Howdy!

    String str = "uppercase me!";
    Supplier<String> makeUppercase = () -> str.toUpperCase();               // (2)
    System.out.println(makeUppercase.get());             // Prints: UPPERCASE ME!

    // Pseudorandom number generator captured and used in lambda expressions: (3)
    Random numGen = new Random();

    // Generate a number between 0 (inclusive) and 100 (exclusive):            (4)
    Supplier<Integer> intSupplier = () -> numGen.nextInt(100);
    System.out.println(intSupplier.get());        // Prints a number in [0, 100).

    // Build a list of Integers with values between 0 (incl.) and 100 (excl.): (5)
    List<Integer> intRefList = listBuilder(5, () -> numGen.nextInt(100));
    System.out.println(intRefList);

    // Build a list of StringBuilders:                                         (6)
    List<StringBuilder> stringbuilderList = listBuilder(6,
        () -> new StringBuilder("str" + numGen.nextInt(10)));          // [0, 10)
    System.out.println(stringbuilderList);

    // Build a list that has the same string:                                  (7)
    List<String> stringList2 = listBuilder(4, () -> "Mini me");
    System.out.println(stringList2);
    // Build a list of LocalTime:                                              (8)
    List<LocalTime> dateList1 = listBuilder(3, () -> LocalTime.now());
    System.out.println(dateList1);

    // Generate a number between 0 (inclusive) and 100 (exclusive):            (9)
    IntSupplier intSupplier2 = () -> numGen.nextInt(100);
    System.out.println(intSupplier2.getAsInt());   // Prints a number in [0, 100).

    // Role many-sided dice:                                                  (10)
    roleDice(6, 100_000, () -> 1 + numGen.nextInt(6));
    roleDice(8, 1_000_000, () -> 1 + (int) (Math.random() * 8));

    // Build an array of doubles with values
    // between 0.0 (incl.) and 5.0 (excl.):                                   (11)
    DoubleSupplier ds = () -> Math.random() * 5;                     // [0.0, 5.0)
    double[] dArray = new double[4];
    for (int i = 0; i < dArray.length; i++) {
      dArray[i] = ds.getAsDouble();
    }
    System.out.println(Arrays.toString(dArray));
  }

  /**
   * Creates a list whose elements are supplied by a Supplier<T>.
   * @param num       Number of elements to put in the list.
   * @param supplier  Supplier that supplies a value to put in the list
   * @return          List created by the method
   */
  public static <T> List<T> listBuilder(int num, Supplier<T> supplier) {   // (12)
    List<T> list = new ArrayList<>();
    for (int i = 0; i < num; ++i) {
      list.add(supplier.get());                                            // (13)
    }
    return list;
  }

  /**
   * Print statistics of rolling a many-sided dice the specified              (14)
   * number of times using an IntSupplier as dice roller.
   */
  public static void roleDice(int numOfSides, int numOfTimes,
                              IntSupplier diceRoller) {
    int[] frequency = new int[numOfSides + 1];         // frequency[0] is ignored.
    for (int i = 0; i < numOfTimes; i++) {
      ++frequency[diceRoller.getAsInt()];                                  // (15)
    }
    System.out.println(Arrays.toString(frequency));
  }
}
```

```
Howdy!
UPPERCASE ME!
83
[15, 24, 48, 3, 16]
```

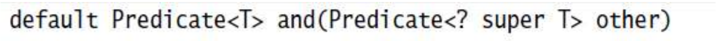
```
[str8, str0, str6, str8, str6, str7]
[Mini me, Mini me, Mini me, Mini me]
[15:32:05.707, 15:32:05.707, 15:32:05.707]
54
[0, 16747, 16723, 16701, 16607, 16637, 16585]
[0, 124918, 124385, 125038, 125451, 124618, 124600, 125230, 125760]
[0.2129971975975531, 0.6933477140020566, 1.3559818256541756, 1.183773498854187]
```

```
Predicate<String> isPalindrome
    = str -> new StringBuilder(str).reverse().toString().equalsIgnoreCase(str);
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isPalindrome);          // Remove all palindromes.
// After:  [Alyah, Java]
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(new Predicate<String>() {
  public boolean test(String str) {
    return new StringBuilder(str).reverse().toString().equalsIgnoreCase(str);
  }
});
// After:  [Alyah, Java]
```

```
Predicate<String> isEvenLen = str -> str.length() % 2 == 0;
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(isEvenLen);             // Remove all even length words.
// After:  [ADA, Alyah, Bob]
```

```
Predicate<String> startsWithA = str -> str.startsWith("A");
// Before: [Otto, ADA, Alyah, Bob, HannaH, Java]
words.removeIf(startsWithA);           // Remove all words that start with "A".
// After:  [Otto, Bob, HannaH, Java]
```

```
default Predicate<T> negate()
```



```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
```

```
static <T> Predicate<T> not(Predicate<? super T> target)
static <T> Predicate<T> isEqual(Object targetRef)
```

```
  ((x.or(y)).and(z))
= ((true.or(y)).and(z))
= (true.and(z))
= (true.and(false))
= false
```

```
// A string that is not a palindrome.
Predicate<String> isNotPalindrome = isPalindrome.negate();      // (1)

// A string with even length or starts with an 'A', and is not a palindrome.
Predicate<String> composedPredicate
    = isEvenLen.or(startsWithA).and(isNotPalindrome);           // (2)
System.out.println("Using composed predicate on \"Adda\": "
        + composedPredicate.test("Adda"));                      // (3) false.

// A string with even length, or it starts with an 'A' and is not a palindrome.
Predicate<String> conditionalOperators
    = str -> str.length() % 2 == 0 || str.startsWith("A")       // (4)
          && !(new StringBuilder(str).reverse().toString().equalsIgnoreCase(str));
System.out.println("Using conditional operators on \"Adda\": "
        + conditionalOperators.test("Adda"));                   // (5) true.
```

```
Predicate<String> isEqualToTarget = Predicate.isEqual("Ada");
System.out.println(isEqualToTarget.test("Adda"));           // false.
System.out.println(Predicate.isEqual("Ada").test("Ada"));   // true.
System.out.println(Predicate.isEqual("null").test("null")); // true.
```

```
Predicate<Integer> isEven = i -> i % 2 == 0;          // Operand unboxed.
System.out.println("2021 is an even number: "
                  + isEven.test(2021));               // Argument boxed. false.

IntPredicate isEvenInt = i -> i % 2 == 0;             // No unboxing.
System.out.println("2021 is an even number: "
                  + isEvenInt.test(2021));            // No boxing. false.
```

```
IntPredicate isOddInt = isEvenInt.negate();          // Negating a predicate.
System.out.println("2020 is an odd number: "
                  + isOddInt.test(2020));            // false.

IntPredicate isInRange = i -> -100 <= i && i <= 100; // Range: [-100, 100]
System.out.println("21 is in range and odd: "
                 + isInRange.and(isOddInt).test(21));// true.
```

```
BiPredicate<String, List<String>> isMember
    = (element, list) -> list.contains(element);
System.out.println(isMember.test("X-File4.doc", filenames));  // true.
```

```
BiPredicate<String, Set<String>> selector = (filename, extensions) ->
    extensions.contains(filename.substring(filename.lastIndexOf('.')));
System.out.println(selector.test("Y-File.pdf", extSet));      // true.
```

```
int number = 21;
BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
System.out.println(isProduct.or(isSum).test(7, 3));       // true.
```

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.BiPredicate;

public class BiPredicateClient {

  public static void main(String[] args) {

    // List with filenames:
    List<String> filenames = new ArrayList<>();
    filenames.add("X-File1.pdf"); filenames.add("X-File2.exe");
    filenames.add("X-File3.fm"); filenames.add("X-File4.doc");
    filenames.add("X-File5.jpg"); filenames.add("X-File6.jpg");
    System.out.println("Filenames: " + filenames);

    // BiPredicate for membership in a list.
    BiPredicate<String, List<String>> isMember =
        (element, list) -> list.contains(element);
    System.out.println(isMember.test("X-File4.doc", filenames));  // true.
    // Set with file extensions:
    Set<String> extSet = new HashSet<>();
    extSet.add(".pdf"); extSet.add(".jpg");
    System.out.println("Required extensions: " + extSet);

    // BiPredicate to determine if a filename has an extension from a specified
    // set of file extensions.
    BiPredicate<String, Set<String>> selector = (filename, extensions) ->
        extensions.contains(filename.substring(filename.lastIndexOf('.')));
    System.out.println(selector.test("Y-File.pdf", extSet));      // true.

    List<String> result = filterList(filenames, extSet, selector);
    System.out.println("Files with required extensions: " + result);

    int number = 21;
    BiPredicate<Integer, Integer> isProduct = (i, j) -> i * j == number;
    BiPredicate<Integer, Integer> isSum     = (i, j) -> i + j == number;
    System.out.println(isProduct.or(isSum).test(7, 3));
  }

  /**
   * Filters a list according to the criteria of the selector.
   * @param list       List to filter
   * @param extSet     Set of file extensions
   * @param selector   BiPredicate that provides the criteria for filtering
   * @return           List of elements that match the criteria
   */
  public static <E, F> List<E> filterList(List<E> list,
                                          Set<F> extSet,
                                          BiPredicate<E, Set<F>> selector) {
    List<E> result = new ArrayList<>();
    for (E element : list)
      if (selector.test(element, extSet))
        result.add(element);
    return result;
  }
}
```

```
Filenames: [X-File1.pdf, X-File2.exe, X-File3.fm, X-File4.doc, X-File5.jpg, X-
File6.jpg]
true
Required extensions: [.pdf, .jpg]
true
Files with required extensions: [X-File1.pdf, X-File5.jpg, X-File6.jpg]
true
```

```
Consumer<Double> formatter = d -> System.out.printf("Value: %.2f%n", d);
formatter.accept(3.145);                       // Value: 3.15
```

```
StringBuilder sb1 = new StringBuilder("Banana");
Consumer<StringBuilder> resizeSB = sb -> sb.setLength(4);
resizeSB.accept(sb1);                              // Bana
```

```
Consumer<StringBuilder> reverseSB = sb -> sb.reverse();
reverseSB.accept(sb1);                       // anaB

Consumer<StringBuilder> printSB
    = sb -> System.out.println("StringBuilder: " + sb);
printSB.accept(sb1);                         // StringBuilder: anaB
```

```
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> System.out.print(s.toLowerCase() + " "));
// otto ada alya bob hannah java
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(new Consumer<String>() {
  public void accept(String s) {
    System.out.print(s.toLowerCase() + " ");
  }
});
// otto ada alya bob hannah java
// [Otto, ADA, Alya, Bob, HannaH, Java]
words.forEach(s -> {if (s.length() % 2 == 0) System.out.print(s + " ");});
// Otto Alya HannaH Java
```

```
resizeSB.andThen(reverseSB)
    .andThen(printSB).accept(new StringBuilder("Banana")); // StringBuilder: anaB
```

```
default Consumer<T> andThen(Consumer<? super T> after)
```

```
IntConsumer sqrt = i -> System.out.printf("%.2f%n", Math.sqrt(i));
IntConsumer sqr = i -> System.out.printf("%d%n", i * i);
sqrt.andThen(sqr).accept(15);    // 3.87
                                 // 225
```

```
BiConsumer<String, Double> formatPrinter
    = (format, obj) -> System.out.printf(format, obj);
formatPrinter.accept("Math.PI:|%10.3f|%n", Math.PI); // Math.PI:|     3.142|
// Map entries (default format): {Java=4, Bob=3, Otto=4, HannaH=6, Alya=4, ADA=3}
strLenMap.forEach((key, value) -> System.out.printf("(%s:%d) ", key, value));
// (Java:4) (Bob:3) (Otto:4) (HannaH:6) (Alya:4) (ADA:3)
```

```
ObjIntConsumer<StringBuilder> resizeSB2 = (sb, len) -> sb.setLength(len);
StringBuilder sb2 = new StringBuilder("bananarama");
resizeSB2.accept(sb2, 6);      // The required length passed as a parameter.
System.out.println("StringBuilder resized: " + sb2);
// StringBuilder resized: banana
```

```
Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
System.out.println("Boolean expression is: " + boolExpr.apply(99));
// Boolean expression is: true

Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
// 24mi = 38.40km
```

```
String[] strArray = {"One", "Two", "Three", "Four"};
List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(s));
System.out.println("Build StringBuilder list: " + sbList);
// Build StringBuilder list: [One, Two, Three, Four]
```

```
listBuilder(String[], String -> StringBuilder)
```

```
List<Integer> intList = listBuilder(strArray, s -> s.length());
System.out.println("Build Integer list: " + intList);
// Build Integer list: [3, 3, 5, 4]
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

public class FunctionClient {
  public static void main(String[] args) {

    // Examples of Function<T,R>:
    Function<Integer, Boolean> boolExpr = i -> 50 <= i && i < 100;
    System.out.println("Boolean expression is: " + boolExpr.apply(99));
    // Boolean expression is: true

    Function<Integer, Double> milesToKms = miles -> 1.6 * miles;
    System.out.printf("%dmi = %.2fkm%n", 24, milesToKms.apply(24));
    // 24mi = 38.40km

    // Create a list of StringBuilders from an array of Strings.
    String[] strArray = {"One", "Two", "Three", "Four"};
    List<StringBuilder> sbList = listBuilder(strArray, s -> new StringBuilder(s));
    System.out.println("Build StringBuilder list: " + sbList);
    // Build StringBuilder list: [One, Two, Three, Four]

    // Create a list of Integers from an array of Strings.
    List<Integer> intList = listBuilder(strArray, s -> s.length());
    System.out.println("Build Integer list: " + intList);
    // Build Integer list: [3, 3, 5, 4]

    /* Composing unary functions. */
    Function<String, String> f = s -> s + "-One";    // (1)
    Function<String, String> g = s -> s + "-Two";    // (2)

    // Using compose() and andThen() methods.
    System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
    System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
    System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One
    System.out.println();

    System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
    System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
    System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
    System.out.println();

    // Examples of primitive unary functions.
    IntFunction<String> intToStr = i -> Integer.toString(i);
    System.out.println(intToStr.apply(2021));        // 2021
    ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
    System.out.println(strToInt.applyAsInt("2021")); // 2021

    IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
    System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                      37, celsiusToFahrenheit.applyAsDouble(37));
    // 37 Celsius = 98.6 Fahrenheit
}

  /**
   * Create a list from an array by applying a Function to each array element.
   * @param arrayT     Array to use for elements
   * @param func       Function to apply to each array element
   * @return           List that is created
   */
  public static <T, R> List<R> listBuilder(T[] arrayT, Function<T, R> func) {
    List<R> listR = new ArrayList<>();
    for (T t : arrayT) {
      listR.add(func.apply(t));
    }
    return listR;
  }
}
```

```
Boolean expression is: true
24mi = 38.40km
Build StringBuilder list: [One, Two, Three, Four]
Build Integer list: [3, 3, 5, 4]
Three-Two-One
Three-Two-One
Three-Two-One

Three-One-Two
Three-One-Two
Three-One-Two

2021
2021
37 Celsius = 98.6 Fahrenheit
```

`f.compose(g).apply(x)` emulates `f.apply(g.apply(x))` or mathematically `f(g(x))`.

`f.andThen(g).apply(x)` emulates `g.apply(f.apply(x))` or mathematically `g(f(x))`.

`f.compose(g).apply(x)` and `g.andThen(f).apply(x)` are equivalent.

```
Function<String, String> f = s -> s + "-One";    // (1)
Function<String, String> g = s -> s + "-Two";    // (2)

System.out.println(f.compose(g).apply("Three")); // (3) Three-Two-One
System.out.println(g.andThen(f).apply("Three")); // (4) Three-Two-One
System.out.println(f.apply(g.apply("Three")));   // (5) Three-Two-One

System.out.println(f.andThen(g).apply("Three")); // (6) Three-One-Two
System.out.println(g.compose(f).apply("Three")); // (7) Three-One-Two
System.out.println(g.apply(f.apply("Three")));   // (8) Three-One-Two
```

```
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
```

```
IntFunction<String> intToStr = i -> Integer.toString(i);
System.out.println(intToStr.apply(2021));        // "2021"
```

```
ToIntFunction<String> strToInt = str -> Integer.parseInt(str);
System.out.println(strToInt.applyAsInt("2021")); // 2021
```

```
IntToDoubleFunction celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
System.out.printf("%d Celsius = %.1f Fahrenheit%n",
                   37, celsiusToFahrenheit.applyAsDouble(37));
// 37 Celsius = 98.6 Fahrenheit
```

```
BiFunction<Double, Double, Double> areaOfRectangle
    = (length, width) -> length * width;            // (Double, Double) -> Double
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

```
BiFunction<String, String, String> concatKeyVal = (key, val) -> key.concat(val);
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(concatKeyVal);
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
// {Dick=silver, Harriet=platinum, Tom=gold}
map.replaceAll(new BiFunction<String, String, String>() {
  public String apply(String key, String val) {
    return key.concat(val);
  }
});
// {Dick=Dicksilver, Harriet=Harrietplatinum, Tom=Tomgold}
```

```
BiFunction<String, String, String> concatStr = (s1, s2) -> s1 + s2;
Function<String, String> postfix1 = s -> s + "nana";
```

```
Function<String, String> postfix2 = s -> s + "s!";
System.out.println(concatStr.andThen(postfix1).andThen(postfix2)
                   .apply("I am going", " ba"));         // I am going bananas!
```

```
BiFunction<String, String, String> concatTwice
              = concatStr.andThen(concatStr);          // Compile-time error!
```

```
default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)
```

```
ToIntBiFunction<String, String> addIntStrs
    = (s1, s2) -> Integer.parseInt(s1) + Integer.parseInt(s2);
System.out.println("10 + 20 = " + addIntStrs.applyAsInt("10", "20"));
// 10 + 20 = 30
```

```
UnaryOperator<Double> area = r -> Math.PI * r * r;
System.out.printf("Area of circle, radius %.2f: %.2f%n", 10.0, area.apply(10.0));
// Area of circle, radius 10.00: 314.16

UnaryOperator<Double> milesToKms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 24.0, milesToKms.apply(24.0));
// 24.00mi = 38.40km
```

```
List<String> team = Arrays.asList("Tom", "Dick", "Harriet");
UnaryOperator<String> toUpper = str -> str.toUpperCase();
team.replaceAll(toUpper);     // [TOM, DICK, HARRIET]
```

```
UnaryOperator<String> f = s -> s + "-One";
UnaryOperator<String> g = s -> s + "-Two";
System.out.println(f.compose(g).apply("Three")); // Three-Two-One
System.out.println(f.andThen(g).apply("Three")); // Three-One-Two
```

```
DoubleUnaryOperator celsiusToFahrenheit = celsius -> 1.8 * celsius + 32.0;
System.out.printf("%.1f Celsius = %.1f Fahrenheit%n",
                  25.0, celsiusToFahrenheit.applyAsDouble(25.0));
// 25.0 Celsius = 77.0 Fahrenheit

DoubleUnaryOperator kms = miles -> 1.6 * miles;
System.out.printf("%.2fmi = %.2fkm%n", 25.0, kms.applyAsDouble(25.0));
// 25.00mi = 40.00km
```

```
IntUnaryOperator incrBy1 = i -> i + 1;
IntUnaryOperator multBy2 = i -> i * 2;
System.out.println(incrBy1.compose(multBy2).applyAsInt(4)); // 9
System.out.println(incrBy1.andThen(multBy2).applyAsInt(4)); // 10
```

```
BinaryOperator<Double> areaOfRectangle = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle.apply(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

```
BinaryOperator<String> concatTwo = (s1, s2) -> s1 + s2;
UnaryOperator<String> postfix1 = s -> s + "nana";
UnaryOperator<String> postfix2 = s -> s + "s!";
System.out.println(concatTwo.andThen(postfix1).andThen(postfix2)
                  .apply("I am going", " ba"));     // I am going bananas!
```

```
String maxStr = BinaryOperator.maxBy(String.CASE_INSENSITIVE_ORDER)
                              .apply("aha", "Madonna");             // Madonna
String minStr = BinaryOperator.minBy(String.CASE_INSENSITIVE_ORDER)
                              .apply("aha", "Madonna");             // aha
```

```
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

```
DoubleBinaryOperator areaOfRectangle2 = (length, width) -> length * width;
System.out.printf("%.2f x %.2f = %.2f%n",
                  25.0, 4.0, areaOfRectangle2.applyAsDouble(25.0, 4.0));
// 25.00 x 4.00 = 100.00
```

```
@FunctionalInterface
interface TriFunction<T, U, V, R> {
  R compute(T t, U u, V v);              // (T, U ,V) -> R
}
```

```
// (Double, Double, Double) -> Double
TriFunction<Double, Double, Double, Double> cubeVol = (x, y, z) -> x * y * z;
System.out.println(cubeVol.compute(10.0,  20.0,  30.0));  // 6000.0
```

```
// Step 1:
// Partial application: double -> DoubleFunction<DoubleUnaryOperator>
DoubleFunction<DoubleFunction<DoubleUnaryOperator>> uniFuncA
    = (x -> (y -> (z -> x * y * z)));              // (1)

// Step 2:
// Partial application: double -> DoubleUnaryOperator
DoubleFunction<DoubleUnaryOperator> uniFuncB
    = uniFuncA.apply(10.0);                        // 10.0 * y * z

// Step 3:
// Partial application: double -> double
DoubleUnaryOperator uniOpC = uniFuncB.apply(20.0); // 10.0 * 20.0 * z
```

```
// Step 4:
// Application:
double vol1 = uniOpC.applyAsDouble(30.0);      // (2) 10.0 * 20.0 * 30.0 = 6000.0
double vol2 = uniFuncA.apply(10.0).apply(20.0).applyAsDouble(30.0); // (3) 6000.0
```

```
// String -> void
Consumer<String> outLE = obj -> System.out.println(obj); // (1a)
Consumer<String> outMR = System.out::println;            // (1b)
outMR.accept("Save trees!");                             // (2)
// Calls System.out.println("Save trees!") that prints: Save trees!
```

```
obj -> System.out.println(obj)
```

```
Supplier<LocalDate> dateNowLE = () -> LocalDate.now();  // (1a) Lambda expression
```

```
Supplier<LocalDate> dateNowMR = LocalDate::now;         // (1b) Method reference
```

```
LocalDate today = datenowMR.get();    // (2) Method reference at (1b) executed.
System.out.println(today.format(DateTimeFormatter.ISO_DATE)); // 2021-03-01
```

```
(args) -> RefType.staticMethod(args)
```

```
RefType::staticMethod
```

```
// String -> Integer
Function<String, Integer> strToIntLE = s -> Integer.parseInt(s); // (3a)
Function<String, Integer> strToIntMR = Integer::parseInt;        // (3b)
```

```
System.out.println(strToIntMR.apply("100"));                     // (4)
// Calls Integer.parseInt("100") that returns the int value 100 which is boxed
// into an Integer.
// (double, double) -> double
DoubleBinaryOperator minDoubleLE = (x, y) -> Math.min(x, y);
DoubleBinaryOperator minDoubleMR = Math::min;          // (5)
System.out.println(minDoubleMR.applyAsDouble(20.0, 30.0));
// Calls Math.min(20.0, 30.0) that returns the double value 20.0.

// (int, int) -> (int)
IntBinaryOperator minIntLE = (x, y) -> Math.min(y, y);
IntBinaryOperator minIntMR = Math::min;                // (6)
System.out.println(minIntMR.applyAsInt(20, 30));
// Calls Math.min(20, 30) that returns the int value 20.
```

```
StringBuilder sb = new StringBuilder("!em esreveR");         // (1)
// () -> StringBuilder
Supplier<StringBuilder> sbReverserLE = () -> sb.reverse();   // (2a)
Supplier<StringBuilder> sbReverserMR = sb::reverse;          // (2b)
System.out.println(sbReverserMR.get());                      // (3)
// Calls sb.reverse() that returns the StringBuilder with character sequence
// "Reverse me!".
```

```
(args) -> expr.instanceMethod(args)
```

```
words.forEach(obj -> System.out.println(obj));
words.forEach(System.out::println);
```

```
String str = "Java Jive";                                          // (4)
// (String, String) -> String
```

```
BinaryOperator<String> replaceOpLE = (s1, s2) -> str.replace(s1, s2); // (5a)
BinaryOperator<String> replaceOpMR = str::replace;                    // (5b)
System.out.println(replaceOpMR.apply("Jive", "Jam"));                 // (6)
// Calls str.replace("Jive", "Jam") that returns the string "Java Jam".
```

```
Predicate<String> p1 = s -> this.equals(s);       // String -> boolean
Predicate<String> p2 = this::equals;              // String -> boolean
Supplier<String> s1 = () -> super.toString();     // () -> String
Supplier<String> s2 = super::toString;            // () -> String
```

```
(arg0, rest) -> arg0.instanceMethod(rest)
```

```
// String -> int
ToIntFunction<String> lenLE = s -> s.length();
ToIntFunction<String> lenMR = String::length;                      // (1)
System.out.println(lenMR.applyAsInt("Java"));                      // 4
// Calls "Java".length() that returns the int value 4.
```

```
List<Integer> intList1 = listBuilder(strArray, s -> s.length()); // Lambda expr.
List<Integer> intList2 = listBuilder(strArray, String::length);  // Method ref.
```

```
// (String, String) -> String
BinaryOperator<String> concatOpLE = (s1, s2) -> s1.concat(s2);
BinaryOperator<String> concatOpMR = String::concat;          // (2)
System.out.println(concatOpMR.apply("Java", " Jive"));       // Java Jive
// Calls "Java".concat(" Jive") that returns the string "Java Jive".
// (List<String>, String) -> boolean
BiPredicate<List<String>, String> containsLE
    = (list, element) -> list.contains(element);
BiPredicate<List<String>, String> containsMR1 = List::contains;          // (3)
BiPredicate<List<String>, String> containsMR2 = List<String>::contains;  // (4)
System.out.println(containsMR2.test(words, "BOB"));  // words is a List<String>.
// Calls words.contains("BOB") that returns a boolean value.
// () -> StringBuilder
Supplier<StringBuilder> sbLE = () -> new StringBuilder();
Supplier<StringBuilder> sbCR = StringBuilder::new;               // (1)
StringBuilder sbRef = sbCR.get();
// Calls new StringBuilder() to create an empty StringBuilder instance.
// String -> StringBuilder
Function<String, StringBuilder> sb3 = s -> new StringBuilder(s);
Function<String, StringBuilder> sb4 = StringBuilder::new;        // (2)
System.out.println(sb4.apply("Build me!"));                      // Build me!
// Calls new StringBuilder("Build me!") to create a StringBuilder instance
// based on the string "Build me!".
// (String, String) -> Locale
BiFunction<String, String, Locale> locConsLE
    = (language, country) -> new Locale(language, country);
BiFunction<String, String, Locale> locConsCR = Locale::new;
System.out.println(locConsCR.apply("en","US"));                  // en_US
// Calls new Locale("en", "US") to create a Locale instance with the specified
// parameter values.
```

```
arg -> new ElementType[arg][]...[]
```

```
// int -> int[]
IntFunction<int[]> intArrConsLE = n -> new int[n];
IntFunction<int[]> intArrConsCR = int[]::new;                      // (1)
int[] intArr = intArrConsCR.apply(4);                              // (2)
// Creates an int array of length 4.
// (int, int) -> int[][]
BiFunction<Integer, Integer, int[][]> twoDimArrConsLE1 = (n, m) -> new int[n][m];
// BiFunction<Integer, Integer, int[][]> twoDimArrConsCR1
//      = int[][]:: new;                                // (3) Compile-time error!
// int -> int[][]
IntFunction<int[][]> twoDimArrConsLE = n -> new int[n][];
IntFunction<int[][]> twoDimArrConsCR = int[][]::new;            // (4)
int[][] twoDimIntArr1 = twoDimArrConsCR.apply(3);               // (5)
// [null, null, null]
for (int i = 0; i < twoDimIntArr1.length; ++i)
  twoDimIntArr1[i] = intArrConsCR.apply(i+1);                   // (6) Calls (1).
// [[0], [0, 0], [0, 0, 0]]
// int -> StringBuilder[]
IntFunction<StringBuilder[]> sbaConsLE = n -> new StringBuilder[n];
IntFunction<StringBuilder[]> sbaConsCR = StringBuilder[]::new;
StringBuilder[] sbArr2 = sbaConsCR.apply(5);                            // (7)
// [null, null, null, null, null]
```

```
public static <A> A[] createArray(int length, IntFunction<A[]> creator) {
//A[] arr = new A[length];       // (8) Cannot create generic array!
  return creator.apply(length);  // Lambda expression or
                                 // array constructor reference executed.
}
```

```
// n -> String[]
String[] strArrLE = createArray(5, n -> new String[n]); // (9)
String[] strArrACE = createArray(5, String[]::new);     // (10)
```

```
DoubleFunction cToF = x -> 1.8 * x + 32.0;            // double -> double
ToIntFunction<String> lenFunc1 = s -> s.length();     // String -> int
ToIntFunction<String> lenFunc2 = String::length;      // String -> int
lenFunc1 = s -> Integer.parseInt(s);                  // String -> int
lenFunc2 = Integer::parseInt;                         // String -> int
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.forEach(i -> System.out.println(i));   // Target type: Consumer<Integer>
numbers.forEach(System.out::println);
```

```
static IntUnaryOperator createLinearFormula(int a, int b) {
  return x -> a * x + b;      // int -> int
}

// Client code:
IntUnaryOperator y = createLinearFormula(10, 5);  // 10 * x + 5
y.applyAsInt(2);                                  // 25
```

```
int ii = 10;
IntUnaryOperator iFunc1 = ii % 2 == 0 ? i -> i * 2 : j -> j + 1;   // int -> int
iFunc1.applyAsInt(4);                                              // 8
//IntUnaryOperator iFunc2 = ii % 2 == 0 ? i -> i * 2
//                          : s -> Integer.parseInt(s);   // Compile-time error!
```

```
System.out.println(((IntUnaryOperator) i -> i * 2).applyAsInt(10));        // 20
System.out.println(((DoubleUnaryOperator) i -> i * 2).applyAsDouble(10.0));// 20.0
```

```
// Object obj1 = StringBuilder::new;                 // Compile-time error!
Object obj2 = (Function<String, StringBuilder>) StringBuilder::new;
Object uFunc1 = (IntUnaryOperator) i -> i * 2;
if (uFunc1 instanceof DoubleUnaryOperator) {                    // (1) false
  DoubleUnaryOperator uFunc2 = (DoubleUnaryOperator) uFunc1;     // (2)
  uFunc2.applyAsDouble(10.0);                                   // (3)
}
```

```
Supplier<Supplier<String>> f = () -> () -> "Hi";
```

```
import java.util.*;
public class Test13RQ5 {
  public static void main(String[] args) {
    List<String> values = new ArrayList<>(List.of("ONE","TWO","THREE","FOUR"));
    values.removeIf(s -> s.length() == 3);
    int sum = 0;
    for (String value: values) {
      sum += value.length();
    }
    System.out.println(sum);
  }
}
```

```
import java.util.*;
public class Test13RQ6 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    values.removeIf(s -> s.toLowerCase().startsWith("a"));
    System.out.println(values);
  }
}
```

```
import java.util.*;
public class Test13RQ8 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("ANNA","JANE","ALICE","JOHN"));
    String s = values.get(0).substring(0,1);
    values.removeIf(s -> s.toLowerCase().startsWith(s));
    values.forEach(s -> System.out.print(s + " "));
  }
}
```

```
import java.util.*;
import java.util.function.*;
public class Test13RQ9 {
  public static void main(String[] args) {
    List<String> values
        = new ArrayList<>(List.of("PLOT","FLOP","LOOP","LEAP"));
    Predicate<String> filter1 = s -> s.contains("O");
    Predicate<String> filter2 = s -> s.endsWith("P");
    values.removeIf(filter1.and(filter2).negate());
    System.out.println(values);
  }
}
```

```
import java.util.*;
import java.util.function.*;
public class Test13RQ10 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
    UnaryOperator<String> f1 = v1 -> v1.substring(0,1).toUpperCase();
    UnaryOperator<String> f2 = v2 -> v2.substring(1).toLowerCase();
    UnaryOperator<String> f3 = f1.compose(f2);
    values.replaceAll(f3);
    System.out.println(values);
  }
}
```

```
import java.util.*;
import java.util.function.*;
public class Test13RQ11 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("ALICE","BOB","JOHN","JANE");
    UnaryOperator<String> f1 = v -> v.toLowerCase();
    values.replaceAll(f1);
    Consumer<String> c1 = s -> s = s.substring(0,1).toUpperCase();
    Consumer<String> c2 = s -> System.out.print(s + " ");
    values.forEach(c1.andThen(c2));
  }
}
```

```
import java.util.*;
class Test {
  private List<Integer> values = new ArrayList<>(List.of(1,2,3,4,5));
  public List<Integer> getValues()        { return values; }
  public static boolean isEven(int value) { return value % 2 != 0; }
  public void printValue(int value)       { System.out.print(value + " "); }
}

public class Main {
  public static void main(String[] args) {
    Test test = new Test();
    test.getValues().removeIf(v -> v % 2 != 0);                // (1)
    test.getValues().forEach(v -> System.out.print(v + " ") ); // (2)
  }
}
```

```
import java.util.function.BiFunction;
public class Test24RQ6 {
  public static void main(String[] args) {
    BiFunction divide = (x, y) -> x/y;
    System.out.print(divide.apply(0.0,0));
  }
}
```

```
import java.util.function.Function;
public class Test13RQ19 {
  public static void main(String[] args) {
    Function f1 = (x) -> "<" + x;
    Function f2 = (x) -> x + ">";
    System.out.print(f2.compose(f1).apply(42));
  }
}
```

```
static boolean equals(Object obj1, Object obj2)
static int hash(Object... values)
static int hashCode(Object obj)
static <T> int compare(T t1, T t2, Comparator<? super T> cmp)
```

```
public boolean equals(Object obj)
```

```
/** A simple version number class */
public class SimpleVNO {

  private int release;
  private int revision;
  private int patch;
  public SimpleVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }
}
```

```
import static java.lang.System.out;
import static java.lang.System.out;

public class TestSimpleVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(SimpleVNO.class);

    // Three individual version numbers.
    SimpleVNO svno1 = new SimpleVNO(9,1,1);                         // (1)
    SimpleVNO svno2 = new SimpleVNO(9,1,1);                         // (2)
    SimpleVNO svno3 = new SimpleVNO(6,6,6);                         // (3)

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
    out.println("Test object reference equality:");                 // (4)
    out.println("  svno1 == svno2:      " + (svno1 == svno2));     // (5)
    out.println("  svno1 == svno3:      " + (svno1 == svno3));     // (6)
    out.println("Test object value equality:");
    out.println("  svno1.equals(svno2): " + svno1.equals(svno2));  // (7)
    out.println("  svno1.equals(svno3): " + svno1.equals(svno3));  // (8)
  }
}
```

```
class SimpleVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): false
  svno1.equals(svno3): false
```

```
return (this == argumentObj)
    || (/* Other criteria for equality. */);
```

```
return super.equals(argumentObj) && compareSubclassSpecificAspects();
```

```
if (!(argumentObj instanceof MyRefType other))
  return false;
// Local variable other can be used in further implementation.
```

```
return (this == argumentObj)                          // Reflexivity test
    || (argumentObj instanceof MyRefType other        // null comparison & cast
        && /* Can use variable other to compare individual fields. */);
```

```
import java.util.Objects;

// Overrides equals(), but not hashCode().
public class UsableVNO {

  private int release;
  private int revision;
  private int patch;
  public UsableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {      // (1)
    return (this == obj)                             // (2)
        || (obj instanceof UsableVNO vno             // (3)
            && this.patch    == vno.patch            // (4)
            && this.revision == vno.revision
            && this.release  == vno.release);
  }
}
```

```
public boolean equals(Object obj)             // (1)
public boolean equals(MyRefType obj)      // Overloaded.
```

```
MyRefType ref1 = new MyRefType();
MyRefType ref2 = new MyRefType();
Object    ref3 = ref2;
boolean b1 = ref1.equals(ref2);    // True. Calls equals() in MyRefType.
boolean b2 = ref1.equals(ref3);    // Always false. Calls equals() in Object.
```

```
return (this == obj)                                      // (2)
    || (/* */);
```

```
return (this == obj)                                    // (2)
    || (obj instanceof UsableVNO vno                    // (3)
        && /* Compare individual fields */);
```

```
if ((obj == null) || (obj.getClass() != this.getClass()))
  return false;
return (this == obj)                                          // (2)
    || (obj instanceof UsableVNO vno                          // (3)
        && this.patch    == vno.patch                         // (4)
        && this.revision == vno.revision
        && this.release  == vno.release);
```

```
(this.productInfo != null && this.productInfo.equals(vno.productInfo)))
```

```
Objects.equals(this.productInfo, vno.productInfo)
```

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(UsableVNO.class);

    // Three individual version numbers.
    UsableVNO svno1 = new UsableVNO(9,1,1);                               // (1)
    UsableVNO svno2 = new UsableVNO(9,1,1);                               // (2)
    UsableVNO svno3 = new UsableVNO(6,6,6);                               // (3)

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                             // (4)
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
    out.println("Test object reference equality:");                       // (5)
    out.println("  svno1 == svno2:      " + (svno1 == svno2));            // (6)
    out.println("  svno1 == svno3:      " + (svno1 == svno3));            // (7)
    out.println("Test object value equality:");
    out.println("  svno1.equals(svno2): " + svno1.equals(svno2));         // (8)
    out.println("  svno1.equals(svno3): " + svno1.equals(svno3));         // (9)
    out.println();

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                           // (10)

    // Searching in an array:
    boolean found = false;
    for (UsableVNO version : versions) {
      found = searchKey.equals(version);                                  // (11)
      if (found) break;
    }
    out.println("Array: " + Arrays.toString(versions));                   // (12)
    out.printf("  Search key %s found in array:    %s%n%n",               // (13)
                searchKey, found);

    // Searching in a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                    // (14)
    out.println("List:  " + vnoList);
    out.printf("  Search key %s contained in list: %s%n%n", searchKey,
                vnoList.contains(searchKey));                             // (15)
  }
}
```

```
class UsableVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): true
  svno1.equals(svno3): false

Array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) found in array:    true

List:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) contained in list: true
```

```
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
```

```
Search key (9.1.1) contained in set:  false
Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO2 {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(UsableVNO.class);

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                           // (1)
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                         // (2)

    // Create a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                  // (3)

    // Searching in a set:
    Set<UsableVNO> vnoSet = new HashSet<>(vnoList);                     // (4)
    out.println("Set: " + vnoSet);
    out.printf("Search key %s contained in set:  %s%n", searchKey,
                vnoSet.contains(searchKey));                            // (5)
    out.println();

    // Search key and its hash code:
    out.printf("Search key %s has hash code: %d%n", searchKey,
               searchKey.hashCode());                                   // (6)

    // Hash values for elements:
    out.println("Hash codes for the elements:");
    for (UsableVNO element : versions) {                                // (7)
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
  }
}
```

```
class UsableVNO
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
```

```
Search key (9.1.1) contained in set:  false

Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

$$\texttt{hashValue} = 11 * 31^{3} + \texttt{release} * 31^{2} + \texttt{revision} * 31^{1} + \texttt{patch}$$

```
import java.util.Objects;

// Overrides both equals() and hashCode().
public class ReliableVNO {

  private int release;
  private int revision;
  private int patch;

  public ReliableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                  // (1)
    return (this == obj)                                          // (2)
        || (obj instanceof ReliableVNO vno                        // (3)
            && this.patch    == vno.patch                         // (4)
            && this.revision == vno.revision
            && this.release  == vno.release);
  }

  @Override public int hashCode() {                               // (5)
    int hashValue = 11;
    hashValue = 31 * hashValue + release;
    hashValue = 31 * hashValue + revision;
    hashValue = 31 * hashValue + patch;
    return hashValue;

//  return Objects.hash(this.release, this.revision, this.patch); // (6)
  }
}
```

```
public int hashCode() {
  int sfVal;
  int hashValue = 11;
  ...
  sfVal = ...   // Compute hash code for each significant field sf. See below.
  hashValue = 31 * hashValue + sfVal;   // (1)
```

```
    ...
    return hashValue;
}
```

```
sfVal = (int) (sf ^ (sf >>> 32));
```

```
long sfValTemp = Double.doubleToLong(sf);
sfVal = (int) (sfValTemp ^ (sfValTemp >>> 32));
sfVal = (sf == null ? 0 : sf.hashCode());
```

$$\texttt{hashValue} = \texttt{'a'} * 31^{2} + \texttt{'b'} * 31^{1} + \texttt{'c'} * 31^{0} = 97 * 31 * 31 + 98 * 31 + 99 = 96354$$

```
return Objects.hash(this.release, this.revision, this.patch); // (6)
```

```
import static java.lang.System.out;
import java.util.*;

public class TestReliableVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(ReliableVNO.class);
    // An array of version numbers.
    ReliableVNO[] versions =  new ReliableVNO[] {                         // (1)
        new ReliableVNO( 3,49, 1), new ReliableVNO( 8,19,81),
        new ReliableVNO( 2,48,28), new ReliableVNO(10,23,78),
        new ReliableVNO( 9, 1, 1)};

    // Search key and its hash code:
    ReliableVNO searchKey = new ReliableVNO( 9, 1, 1);                     // (2)
    out.printf("Search key %s has hash code: %d%n", searchKey,
        searchKey.hashCode());                                             // (3)

    // Print hash values:
    out.println("Hash codes:");
    for (ReliableVNO element : versions) {                                 // (4)
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
    out.println();

    // Searching in a set:
    Set<ReliableVNO> vnoSet = new HashSet<>(Arrays.asList(versions));      // (5)
    out.println("Set: " + vnoSet);
    out.printf("Search key %s contained in set: %s%n%n", searchKey,
                vnoSet.contains(searchKey));                               // (6)

    // Searching in a map:
    Map<ReliableVNO, Integer> versionStatistics = new HashMap<>();         // (7)
    versionStatistics.put(versions[0], 2000);
    versionStatistics.put(versions[1], 3000);
    versionStatistics.put(versions[2], 4000);
    versionStatistics.put(versions[3], 5000);
    versionStatistics.put(versions[4], 6000);
    out.println("Map: " + versionStatistics);                              // (8)
    out.printf("Search key %s contained in map: %s%n", searchKey,
               versionStatistics.containsKey(searchKey));                  // (9)
  }
}
```

```
class ReliableVNO
Search key (9.1.1) has hash code: 336382
Hash codes:
    (3.49.1): 332104
   (8.19.81): 336059
   (2.48.28): 331139
  (10.23.78): 338102
     (9.1.1): 336382

Set: [(10.23.78), (2.48.28), (9.1.1), (3.49.1), (8.19.81)]
Search key (9.1.1) contained in set: true

Map: {(10.23.78)=5000, (2.48.28)=4000, (9.1.1)=6000, (3.49.1)=2000,
      (8.19.81)=3000}
Search key (9.1.1) contained in map: true
```

```
@Override public boolean equals(Object obj) {
  return (this == obj)
      || (obj instanceof Whatever other
          && this.compareTo(other) == 0);
}
```

```
import java.util.Comparator;
import java.util.Objects;

public final class VersionNumber implements Comparable<VersionNumber> {

  private final int release;
  private final int revision;
  private final int patch;

  public VersionNumber(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                      // (1)
    return (this == obj)                                              // (2)
        || (obj instanceof VersionNumber vno                          // (3)
            && this.patch    == vno.patch
            && this.revision == vno.revision
            && this.release  == vno.release);
  }

  @Override public int hashCode() {
    return Objects.hash(this.release, this.revision, this.patch);   // (4)
  }

  @Override public int compareTo(VersionNumber vno) {               // (5)
    // Compare the release numbers.                                     (6)
    if (this.release != vno.release)
      return Integer.compare(this.release, vno.release);

    // Release numbers are equal,                                       (7)
    // must compare revision numbers.
    if (this.revision != vno.revision)
      return Integer.compare(this.revision, vno.revision);

    // Release and revision numbers are equal,                          (8)
    // patch numbers determine the ordering.
    return Integer.compare(this.patch, vno.patch);
  }

  /*
  @Override public int compareTo(VersionNumber vno) {               // (9a)
```

```
    return Comparator.comparingInt(VersionNumber::getRelease)       // (10a)
                     .thenComparingInt(VersionNumber::getRevision)  // (11a)
                     .thenComparingInt(VersionNumber::getPatch)     // (12a)
                     .compare(this, vno);                           // (13a)
  }
  */
}
```

```
Integer iRef = 10;
Double dRef = 3.14;
String str = "ten";
StringBuilder sb = new StringBuilder("ten");
boolean b1 = iRef.compareTo(str) == 0 ;  // compareTo(Integer) not applicable to
                                         // arguments (String).
boolean b2 = dRef.compareTo(iRef) > 0;   // compareTo(Integer) not applicable to
                                         // arguments (Double).
boolean b3 = sb.compareTo(str) == 0; // compareTo(StringBuilder) not applicable to
                                     // arguments (String).
```

```
public final class VersionNumber implements Comparable<VersionNumber> {
  ...
  @Override public int compareTo(VersionNumber vno) {                // (5)
  ...
  }
  ...
}
```

```
public int compareTo(Object obj) {   // NOT A GOOD IDEA TO RELY ON THIS METHOD!
  return this.compareTo((VersionNumber) obj);
}
```

```
if (this.release != vno.release)
  return Integer.compare(this.release, vno.release);
// Next field comparison
```

```
int releaseDiff = release - vno.release;
if (releaseDiff != 0)
  return releaseDiff;
// Next field comparison
```

```
@Override public int compareTo(VersionNumber vno) {                // (9b)
  Comparator<VersionNumber> c1
      = Comparator.comparingInt(vn -> vn.getRelease());             // (10b)
  Comparator<VersionNumber> c2
      = c1.thenComparingInt(vn -> vn.getRevision());                // (11b)
  Comparator<VersionNumber> c3
      = c2.thenComparingInt(vn -> vn.getPatch());                   // (12b)
//return c3.compare(this, vno);                                     // (13b)
  return Objects.compare(this, vno, c3);                            // (13c)
}
```

```
import static java.lang.System.out;
import java.util.*;

public class TestVersionNumber {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(VersionNumber.class);

    // Create an unsorted array of version numbers:
    VersionNumber[] versions =  new VersionNumber[] {                      // (1)
        new VersionNumber( 3,49, 1), new VersionNumber( 8,19,81),
        new VersionNumber( 2,48,28), new VersionNumber(10,23,78),
        new VersionNumber( 9, 1, 1)};
    out.println("Unsorted array: " + Arrays.toString(versions));

    // Create an unsorted list:
    List<VersionNumber> vnoList = Arrays.asList(versions);                 // (2)
    out.println("Unsorted list:  " + vnoList);
    // Create an unsorted map:
    Map<VersionNumber, Integer> versionStatistics = new HashMap<>();       // (3)
    versionStatistics.put(versions[0], 2000);
    versionStatistics.put(versions[1], 3000);
    versionStatistics.put(versions[2], 4000);
    versionStatistics.put(versions[3], 5000);
    versionStatistics.put(versions[4], 6000);
    out.println("Unsorted map: " + versionStatistics);

    // Sorted set:
    Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                 // (4)
    out.println("Sorted set: " + sortedSet);

    // Sorted map:
    Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics);//(5)
    out.println("Sorted map: " + sortedMap);

    // Sorted list:
    Collections.sort(vnoList);                                             // (6)
    out.println("Sorted list:    " + vnoList);

    // Searching in sorted list:
    VersionNumber searchKey = new VersionNumber( 9, 1, 1);                 // (7)
    int resultIndex = Collections.binarySearch(vnoList, searchKey);        // (8)
    out.printf("Binary search in sorted list found key %s at index: %d%n",
               searchKey, resultIndex);

    // Sorted array:
    Arrays.sort(versions);                                                 // (9)
    out.println("Sorted array:   " + Arrays.toString(versions));

    // Searching in sorted array:
    int resultIndex2 = Arrays.binarySearch(versions, searchKey);           // (10)
    out.printf("Binary search in sorted array found key %s at index: %d%n",
               searchKey, resultIndex2);
  }
}
```

```
class VersionNumber
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted map: {(10.23.78)=5000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
               (2.48.28)=4000}
Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}
Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted list found key (9.1.1) at index: 3
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted array found key (9.1.1) at index: 3
```

```
Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                   // (4)
...
Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics); // (5)
```

```
Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}
```

```
Collections.sort(vnoList);                                               // (6)
```

```
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

```
VersionNumber searchKey = new VersionNumber( 9, 1, 1);                // (7)
int resultIndex = Collections.binarySearch(vnoList, searchKey);        // (8)
```

```
Binary search in sorted list found key (9.1.1) at index: 3
```

```
Arrays.sort(versions);                                                  // (9)
```

```
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

```
int resultIndex2 = Arrays.binarySearch(versions, searchKey);           // (10)
```

```
Binary search in sorted array found key (9.1.1) at index: 3
```

```
import java.util.*;

public class ComparatorUsage {
  public static void main(String[] args) {

    String[] words = {"court", "Stuart", "report", "Resort",            // (1)
                      "assort", "support", "transport", "distort"};

    // Choice of comparator.
    Set<String> strSet1 = new TreeSet<>();              // (1a) Natural ordering
    Set<String> strSet2 = new TreeSet<>(String.CASE_INSENSITIVE_ORDER); // (1b)
    Set<String> strSet3 = new TreeSet<>(                // (1c) Rhyming ordering
      (String obj1, String obj2) -> {
        // Create reversed versions of the strings:                     (2)
        String reverseStr1 = new StringBuilder(obj1).reverse().toString();
        String reverseStr2 = new StringBuilder(obj2).reverse().toString();
        // Compare the reversed strings lexicographically.
        return reverseStr1.compareTo(reverseStr2);                   // (3)
      }
    );
    Set<String> strSet4 = new TreeSet<>(
        Comparator.comparingInt(String::length)         // (4) First length, then by
                  .thenComparing(Comparator.naturalOrder())// (5) natural ordering
    );

    // Add the elements from the words array to a set and print the set:
    Collections.addAll(strSet1, words);                              // (6)
    System.out.println("Natural order:\n" + strSet1);                // (7)
    Collections.addAll(strSet2, words);
    System.out.println("Case insensitive order:\n" + strSet2);
    Collections.addAll(strSet3, words);
    System.out.println("Rhyming order:\n" + strSet3);
    Collections.addAll(strSet4, words);
    System.out.println("Length, then natural order:\n" + strSet4);
  }
}
```

```
Natural order:
[Resort, Stuart, assort, court, distort, report, support, transport]
Case insensitive order:
[assort, court, distort, report, Resort, Stuart, support, transport]
Rhyming order:
[Stuart, report, support, transport, Resort, assort, distort, court]
Length, then natural order:
[court, Resort, Stuart, assort, report, distort, support, transport]
```

```
import static java.lang.System.out;
import java.util.*;

public class UsingVersionNumberComparator {

  public static void main(String[] args) {
    VersionNumber[] versions = new VersionNumber[] {                       // (1)
        new VersionNumber(3, 49, 1), new VersionNumber(8, 19, 81),
        new VersionNumber(2, 48, 28), new VersionNumber(10, 23, 78),
        new VersionNumber(9, 1, 1) };

    List<VersionNumber> vnList = new ArrayList<>();
    Collections.addAll(vnList, versions);                                  // (2)
    out.println("List before sorting:\n  " + vnList);
    Collections.sort(vnList, Comparator.reverseOrder());                   // (3)
    out.println("List after sorting according to " +
                "reverse natural ordering:\n  " + vnList);

    VersionNumber searchKey = new VersionNumber(9, 1, 1);
    int resultIndex = Collections.binarySearch(vnList, searchKey,
                                           Comparator.reverseOrder());  // (4)
    out.printf("Binary search in list using reverse natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);

    resultIndex = Collections.binarySearch(vnList, searchKey);             // (5)
    out.printf("Binary search in list using natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);
  }
}
```

```
List before sorting:
  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
List after sorting according to reverse natural ordering:
  [(10.23.78), (9.1.1), (8.19.81), (3.49.1), (2.48.28)]
Binary search in list using reverse natural ordering found key (9.1.1) at index: 1
Binary search in list using natural ordering found key (9.1.1) at index: -6
```

```
default Comparator<T> reversed()
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
static <T> Comparator<T> nullsFirst(Comparator<? super T> cmp)
static <T> Comparator<T> nullsLast(Comparator<? super T> cmp)
static <T, U> Comparator<T>
        comparing(Function<? super T,? extends U> func,
                  Comparator<? super U> cmp)
```

```
static <T, U extends Comparable<? super U>> Comparator<T>
        comparing(Function<? super T,? extends U> func)
```

```
static <T> Comparator<T>
        comparingPrimType(ToPrimTypeFunction<? super T> func)
```

```
default Comparator<T>
        thenComparing(Comparator<? super T> cmp)
```

```
default <U> Comparator<T>
        thenComparing(Function<? super T,? extends U> func,
                      Comparator<? super U> cmp)
```

```
default <U extends Comparable<? super U>> Comparator<T>
        thenComparing(Function<? super T,? extends U> func)
```

```
default Comparator<T>
        thenComparingPrimType(ToPrimTypeFunction<? super T> func)
```

```
String func(Object x, Object y) {
  return (x == y) + " " + x.equals(y) + " " + (x.hashCode() == y.hashCode());
}
```

```
public class Pair {
  private int a, b;
  public Pair(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public boolean equals(Object o) {
    return (this == o) || (o instanceof Pair) && equalsImpl((Pair) o);
  }

  private boolean equalsImpl(Pair o) {
    // (1) INSERT CODE HERE
  }
}
```

```
import java.util.*;
public class Measurement {
  private int count;
  private int accumulated;
  public Measurement() {}
  public void record(int v) {
    count++;
    accumulated += v;
  }
  public int average() {
    return accumulated/count;
  }
  public boolean equals(Object other) {
    if (this == other)
      return true;
    if (!(other instanceof Measurement))
      return false;
    Measurement o = (Measurement) other;
    if (count != 0 && o.count != 0)
      return average() == o.average();
    return count == o.count;
  }
  public int hashCode() {
    // (1) INSERT CODE HERE
  }
}
```

```
return (count << 16) ^ accumulated;
return count == 0 ? 0 : average();
```

```
import java.util.Comparator;
import java.util.Comparator;
class Person  implements Comparable<Person > {
  private String name;
  private int age;
  Person (String name, int age) { this.name = name; this.age = age; }

  public int compareTo(Person  p2) {
    Comparator<String> strCmp = Person.cmp();
    int status = strCmp.compare(this.name, p2.name);
    if (status == 0) {
      Comparator<Integer> intCmp = Person.cmp();
      status = intCmp.compare(this.age, p2.age);
    }
    return status;
  }

  public static <E extends Comparable<E>> Comparator<E> cmp() {
    return  (e1, e2) -> e2.compareTo(e1);
  }
}

public class Main {
  public static void main(String[] args) {
    Person  p1 = new Person ("Tom", 20);
    Person  p2 = new Person ("Dick", 30);
    Person  p3 = new Person ("Tom", 40);
    System.out.println((p1.compareTo(p2) < 0) + " " + (p1.compareTo(p3) < 0));
  }
}
```

```
import java.util.Comparator;
class CompDecls {

  public static <E extends Comparable<E>> Comparator<E> cmp1() {     // (1)
    return new Comparator<E>() {
      public int compare(E e1, E e2) { return e2.compareTo(e1); }
    };
  }

  public static <E extends Comparable<E>> Comparator<E> cmp2() {     // (2)
    return (e1, e2) -> e2.compareTo(e1);
  }

  public static <E extends Comparable<E>> Comparator<E> cmp3() {     // (3)
    return ((Comparator<E>)(e1, e2) -> e1.compareTo(e2)).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp4() {     // (4)
    return Comparable::compareTo;
  }

  public static <E extends Comparable<E>> Comparator<E> cmp5() {     // (5)
    return ((Comparator<E>)Comparable::compareTo).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp6() {     // (6)
    Comparator<E> cmp =  Comparable::compareTo;
    return cmp.reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp7() {     // (7)
    return Comparator.reverseOrder();
  }
}
```

```
import java.util.*;
public class Test14RQ8 {
  public static void main(String[] args) {
    Integer[] values = {-42,15,-23,19,11,71};
    Arrays.sort(values, (v1, v2) -> v1.toString().compareTo(v2.toString()));
    System.out.println(Arrays.toString(values));
  }
}
```

```
import java.util.*;
import java.util.function.*;
public class Album {
  private static List<Album> albums = new ArrayList<>();
  private String title;
  private Album(String title) { this.title = title; }
  public String toString() { return title; }
  public static void addAlbum(String title) {
    albums.add(new Album(title));
  }
  public static void sortAlbums(Comparator<Album> c) {
    albums.sort(c);
  }
  public static void processAlbums(Consumer<Album> c) {
    albums.forEach(c);
  }
}
```

```
public class Test14RQ11 {
  public static void main(String[] args) {
    Album.addAlbum("New Songs");
    Album.addAlbum("More Songs");
    Album.addAlbum("Greatest Hits");
    Album.addAlbum("Old Songs");
    Album.sortAlbums((a1, a2) -> a1.toString().length() - a2.toString().length());
    Album.processAlbums(a -> {
      System.out.print(a.toString() + " ");
    });
  }
}
```

New Songs Old Songs More Songs Greatest Hits

New Songs More Songs Greatest Hits Old Songs

```
import java.util.Objects;
public class Album1 {
  private String title;
  public Album1(String title) { this.title = title; }
  public int hashCode() {
    int hash = 7;
    hash = 29 * hash + Objects.hashCode(this.title);
    return hash;
  }
  public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj == null) { return false; }
    if (getClass() != obj.getClass()) { return false; }
    final Album1 other = (Album1) obj;
    return Objects.equals(this.title, other.title);
  }
}
```

```
public class LP extends Album1 {
  public LP(String title) { super(title); }
}
```

```
public class Test14RQ9 {
  public static void main(String[] args) {
    Album1 a1 = new Album1("Some Music");
    Album1 a2 = new LP("Some Music");
    if (a1.equals(a2)) {
      System.out.println("Same album");
    } else {
       System.out.println("These are different albums");
    }
  }
}
```

`int size()`	
`boolean isEmpty()`	
`boolean contains(Object element)`	
`boolean add(E element)`	Optional
`boolean remove(Object element)`	Optional

`boolean`	`containsAll(Collection<?> c)`	
`boolean`	`addAll(Collection<? extends E> c)`	Optional
`boolean`	`removeAll(Collection<?> c)`	Optional
`boolean`	`retainAll(Collection<?> c)`	Optional
`void`	`clear()`	Optional

`Iterator<E> iterator()`	From `Iterable<E>` interface.

`default void forEach(Consumer<? super T> action)` From `Iterable<E>` interface.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

public class IterationOverCollections {
  public static void main(String[] args) {

    // Create a collection of names.                                   (1)
    Collection<String> collectionOfNames = new ArrayList<>();
    Collections.addAll(collectionOfNames, "Alice", "Bob", "Tom","Dick","Harriet");
    System.out.println(collectionOfNames);

    // Using an explicit iterator.
    Iterator<String> iterator = collectionOfNames.iterator();        // (2)
    while (iterator.hasNext()) {                                     // (3)
      String name = iterator.next();                                 // (4)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();

    // Using the forEachRemaining() method of the Iterator interface.
    iterator = collectionOfNames.iterator();                         // (5)
    iterator.forEachRemaining(name ->                                // (6)
        System.out.print(name.toUpperCase() + " "));
    System.out.println();

    // Using the for(:) loop on an Iterable.
    for (String name : collectionOfNames) {                          // (7)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();
    // Using the forEach() method of the Collection interface.
    collectionOfNames.forEach(name ->                               // (8)
        System.out.print(name.toUpperCase() + " "));
    System.out.println();

    // Filtering using an explicit iterator.
    iterator = collectionOfNames.iterator();                        // (9)
    while (iterator.hasNext()) {                                    // (10)
      String name = iterator.next();                                // (11)
      if (name.length() == 3)
        iterator.remove();                                          // (12)
    }
    System.out.println(collectionOfNames);

    // Filtering using the removeIf() method of the Collection interface.
    collectionOfNames.removeIf(name -> name.startsWith("A"));       // (13)
        System.out.println(collectionOfNames);

    // Using the stream() method of the Collection interface.
    collectionOfNames.stream()                                      // (14)
                     .forEach(name -> System.out.print(name.toUpperCase() + " "));
    System.out.println();
  }
}
```

```
[Alice, Bob, Tom, Dick, Harriet]
```

```
[Alice, Bob, Tom, Dick, Harriet]
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
[Alice, Dick, Harriet]
[Dick, Harriet]
DICK HARRIET
```

`boolean hasNext()`	From `Iterator<E>` interface.

`E next()`	From `Iterator<E>` interface.

`void remove()`	Optional	From `Iterator<E>` interface.

`default void`
`forEachRemaining(Consumer<? super E> action)` From `Iterator<E>` interface.

```
Iterator<String> iterator = collectionOfNames.iterator();        // (2)
while (iterator.hasNext()) {                                     // (3)
  String name = iterator.next();                                 // (4)
  System.out.print(name.toUpperCase() + " ");
}
```

```
for (Iterator<String> iterator = collectionOfNames.iterator();
     iterator.hasNext(); /* Empty increment. */) {
  String name = iterator.next();
  System.out.print(name.toUpperCase() + " ");
}
```

```
iterator = collectionOfNames.iterator();                                // (5)
iterator.forEachRemaining(name ->                                        // (6)
    System.out.print(name.toUpperCase() + " "));
```

```
for (String name : collectionOfNames) {                                // (7)
  System.out.print(name.toUpperCase() + " ");
}
```

```
collectionOfNames.forEach(name ->
    System.out.print(name.toUpperCase() + " "));                    // (8)
```

```
// [Alice, Bob, Tom, Dick, Harriet]
iterator = collectionOfNames.iterator();                              // (9)
while (iterator.hasNext()) {                                          // (10)
  String name = iterator.next();                                      // (11)
  if (name.length() == 3)
    iterator.remove();                                                // (12)
}
// [Alice, Dick, Harriet]
```

```
default boolean removeIf(Predicate<? super E> filter)
```

```
// [Alice, Dick, Harriet]
collectionOfNames.removeIf(name -> name.startsWith("A"));        // (13)
// [Dick, Harriet]
```

```
default Stream<E> stream()
default Stream<E> parallelStream()
```

```
collectionOfNames.stream()                                   // (14)
                 .forEach(name -> System.out.print(name.toUpperCase() + " "));
```

```
default <T> T[] toArray(IntFunction<T[]> generator)
```

```
IntFunction<String[]> createStrArray = nn -> new String[nn];          // (7)
String[] strArray5 = strSet.toArray(createStrArray);                  // (8a)
String[] strArray6 = strSet.toArray(String[]::new);                   // (8b)
String[] strArray7 = strSet.toArray(createStrArray.apply(0));         // (8c)
```

```
import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;
import java.util.function.IntFunction;

public class CollectionToArray {
  public static void main(String[] args) {

    Collection<String> strSet = new HashSet<>();
    strSet.add("2021"); strSet.add("2022"); strSet.add("2023");
    int n = strSet.size();

    Object[] objects = strSet.toArray();                // (1)
//  String[] string = strSet.toArray();                 // (2) Compile-time error!

    Object[] objArray = strSet.toArray(new Object[0]);                 // (3)
    System.out.println("Array size: " + objArray.length);
    System.out.println("Array type: " + objArray.getClass().getName());
    System.out.println("Actual element type: " +
                       objArray[0].getClass().getName());

    String[] strArray1 = new String[0];
    String[] strArray2 = strSet.toArray(strArray1);                    // (4)
    System.out.println("strArray1 == strArray2: " + (strArray1 == strArray2));

    String[] strArray3 = new String[n];
    String[] strArray4 = strSet.toArray(strArray3);                    // (5)
    System.out.println("strArray3 == strArray4: " + (strArray3 == strArray4));

//  Integer[] intArray = strSet.toArray(new Integer[n]);     // (6) Runtime error!

    IntFunction<String[]> createStrArray = nn -> new String[nn];           // (7)
    String[] strArray5 = strSet.toArray(createStrArray);               // (8a)
    String[] strArray6 = strSet.toArray(String[]::new);                // (8b)
    String[] strArray7 = strSet.toArray(createStrArray.apply(0));      // (8c)
    System.out.println("strArray5: " + Arrays.toString(strArray5));
    System.out.println("strArray6: " + Arrays.toString(strArray6));
    System.out.println("strArray7: " + Arrays.toString(strArray7));
  }
}
```

```
Array size: 3
Array type: [Ljava.lang.Object;
Actual element type: java.lang.String
strArray1 == strArray2: false
strArray3 == strArray4: true
strArray5: [2023, 2022, 2021]
strArray6: [2023, 2022, 2021]
strArray7: [2023, 2022, 2021]
```

```
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
```

```
interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  boolean hasPrevious();
  E next();                 // Element after the cursor
  E previous();             // Element before the cursor
  int nextIndex();          // Index of element after the cursor
  int previousIndex();      // Index of element before the cursor
  void remove();            // Optional
  void set(E o);            // Optional
  void add(E o);            // Optional
}
```

```
// Find the number of digits that were correctly included.                  (3)
List<Integer> duplicate = new ArrayList<>(secretSolution);
duplicate.retainAll(guess);
numOfDigitsIncluded = duplicate.size();
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

public class TakeAGuess {
  static final int NUM_DIGITS = 5;

  public static void main(String[] args) {

    // Sanity check on the given data.
    try {
      if (args.length != 1 || args[0].length() != NUM_DIGITS)
        throw new IllegalArgumentException();
      Integer.parseInt(args[0]);
    } catch(IllegalArgumentException nfe) {
      System.err.println("Guess should be " + NUM_DIGITS + " digits.");
      return;
    }
    String guessStr = args[0];
    System.out.println("Guess: " + guessStr);

    /* Initialize the solution list. This program has a fixed solution: 53272 */
    List<Integer> secretSolution = new ArrayList<>();                       // (1)
    Collections.addAll(secretSolution, 5, 3, 2, 7, 2);

    // Convert the guess from string to a list of Integers.                    (2)
    List<Integer> guess = new ArrayList<>();
    for (int i = 0; i < guessStr.length(); i++)
      guess.add(Character.getNumericValue(guessStr.charAt(i)));

    // Find the number of digits that were correctly included.                 (3)
    List<Integer> duplicate = new ArrayList<>(secretSolution);
    int numOfDigitsIncluded = 0;
    for (int i = 0; i < NUM_DIGITS; i++)
      if (duplicate.remove(guess.get(i))) numOfDigitsIncluded++;

    /* Find the number of digits correctly placed by comparing the two
       lists, element by element, counting each correct placement. */
    // Need two iterators to traverse through the guess and solution lists.    (4)
    ListIterator<Integer> correct = secretSolution.listIterator();
    ListIterator<Integer> attempt = guess.listIterator();
    int numOfDigitsPlaced = 0;
    while (correct.hasNext())
      if (correct.next().equals(attempt.next())) numOfDigitsPlaced++;

    // Print the results.
    System.out.println(numOfDigitsIncluded + " digit(s) correctly included.");
    System.out.println(numOfDigitsPlaced +   " digit(s) correctly placed.");
  }
}
```

```
Guess: 32227
4 digit(s) correctly included.
1 digit(s) correctly placed.
```

```
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5,
                     E e6, E e7, E e8, E e9, E e10)
```

```
@SafeVarargs static <E> Set<E> of(E... elements)
```

```
static <E> Set<E> copyOf(Collection<? extends E> collection)
```

```
Set<String> set = Set.of("Tom", "Dick", "Harriet");
//  set.add("Harry");                           // UnsupportedOperationException
//  set.remove(2);                              // UnsupportedOperationException
System.out.println(set);                        // [Harriet, Tom, Dick]
System.out.println(set instanceof HashSet);     // false
```

```
Set<String> coinSet = Set.of("dime", "nickel", null); // NullPointerException
```

```
Set<Integer> intSet1 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);     // Fixed-arity
Set<Integer> intSet2 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Varargs
System.out.println(intSet1);        // [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
System.out.println(intSet2);        // [7, 6, 5, 4, 3, 2, 1, 11, 10, 9, 8]
```

```
String[] strArray = {"Tom", "Dick", "Harriet"};
Set<String> strSet = Set.of(strArray);                    // (1) Set of String
Set<String[]> strArraySet = Set.<String[]>of(strArray);  // (2) Set of String[]
System.out.println(strSet);                  // [Harriet, Dick, Tom]
System.out.println(strArraySet);             // [[Ljava.lang.String;@3b22cdd0]
```

```
Set<String> fab4 = new HashSet<>();
fab4.add("John"); fab4.add("Paul"); fab4.add("George"); fab4.add("Ringo");
System.out.println(fab4);                    // [George, John, Ringo, Paul]
Set<String> fabAlways = Set.copyOf(fab4);    // (1)
fab4.remove("John"); fab4.remove("George");  // Modify original set
System.out.println(fab4);                    // [Ringo, Paul]
System.out.println(fabAlways);               // [John, Paul, Ringo, George]
```

```
HashSet(Collection<? extends E> c)
HashSet(int initialCapacity, float loadFactor)
```

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
public class IterationHashSetAndLinkedHashSet {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<>();                              // (1)
    set1.add("breakfast"); set1.add("lunch"); set1.add("dinner");
    System.out.println("Serving meals from a HashSet (order can vary):");
    for (String meal : set1) {
      System.out.println(meal);
    }
    Set<String> set2 = new LinkedHashSet<>();                        // (2)
    set2.add("breakfast"); set2.add("lunch"); set2.add("dinner");
    System.out.println("Serving meals from a LinkedHashSet" +
                       " (always insertion order):");
      for (String meal : set2) {
        System.out.println(meal);
      }
    }
  }
```

```
Serving meals from a HashSet (order can vary):
dinner
breakfast
lunch
Serving meals from a LinkedHashSet (always insertion order):
breakfast
lunch
dinner
```

```
// Determine whether a common subset exists:                 (4)
Set<Character> commonSubset = new HashSet<>(encountered);
commonSubset.retainAll(characters);
boolean areDisjunct = commonSubset.size()==0;
```

```
// Determine superset and subset relations.                (5)
boolean isSubset = encountered.containsAll(characters);
boolean isSuperset = characters.containsAll(encountered);
```

```
    encounteredaddAll(characters);                                // (6)
```

```
import java.util.HashSet;
import java.util.Set;

public class CharacterSets {
  public static void main(String[] args) {

    // A set for keeping track of all characters previously encountered.
    Set<Character> encountered = new HashSet<>();               // (1)

    // For each program argument in the command line ...
    for (String argument : args) {

      // Convert the current argument to a set of characters.
      Set<Character> characters = new HashSet<>();              // (2)
      int size = argument.length();
      // For each character in the argument...
      for (int j = 0; j < size; j++)
        // add character to the characters set.
        characters.add(argument.charAt(j));                     // (3)

      // Determine whether a common subset exists:                 (4)
      Set<Character> commonSubset = new HashSet<>(encountered);
      commonSubset.retainAll(characters);
      boolean areDisjunct = commonSubset.size()==0;

      if (areDisjunct) {
        System.out.println(characters + " and " + encountered + " are disjunct.");
      } else {
        // Determine superset and subset relations.                (5)
        boolean isSubset = encountered.containsAll(characters);
        boolean isSuperset = characters.containsAll(encountered);
        if (isSubset && isSuperset)
          System.out.println(characters + " is equivalent to " + encountered);
        else if (isSubset)
          System.out.println(characters + " is a subset of " + encountered);
        else if (isSuperset)
          System.out.println(characters + " is a superset of " + encountered);
        else
          System.out.println(characters + " and " + encountered + " have " +
                             commonSubset + " in common.");
      }

      // Update the set of characters encountered so far.
      encountered.addAll(characters);                           // (6)
```

```
        }
    }
}
```

```
>java CharacterSets i said i am maids
```

```
[i] and [] are disjunct.
[d, a, s, i] is a superset of [i]
[i] is a subset of [d, a, s, i]
[a, m] and [d, a, s, i] have [a] in common.
[d, a, s, m, i] is equivalent to [d, a, s, m, i]
```

```
// Range-view operations
SortedSet<E> headSet(<E> toElement)
SortedSet<E> tailSet(<E> fromElement)
SortedSet<E> subSet(<E> fromElement, <E> toElement)
```

```
// Comparator access
Comparator<? super E> comparator()
```

```
// Range-view operations
NavigableSet<E> headSet(E toElement,   boolean inclusive)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

```
TreeSet(Comparator<? super E> comparator)
TreeSet(Collection<? extends E> collection)
```

```
NavigableSet<String> strSetB = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
Collections.addAll(strSetB, "strictly", "dancing", "Java", "Ballroom");
System.out.println(strSetB);          // [Ballroom, dancing, Java, strictly]
```

```
import java.util.Collections;
import java.util.NavigableSet;
import java.util.TreeSet;
import static java.lang.System.out;

public class SetNavigation {
  public static void main(String[] args) {

    NavigableSet<String> strSetA = new TreeSet<>();                         // (1)
    Collections.addAll(strSetA, "Strictly", "Java", "dancing", "ballroom"); // (2)
    out.println("Before: " + strSetA);      // [Java, Strictly, ballroom, dancing]

    out.println("\nSubset-views:");                   // (3)
    out.println(strSetA.headSet("ballroom", true));  // [Java, Strictly, ballroom]
    out.println(strSetA.headSet("ballroom", false)); // [Java, Strictly]
    out.println(strSetA.tailSet("Strictly", true));  // [Strictly, ballroom,
                                                     //  dancing]
    out.println(strSetA.tailSet("Strictly", false)); // [ballroom, dancing]
    out.println(strSetA.subSet("A", false, "Z", false )); // [Java, Strictly]
    out.println(strSetA.subSet("a", false, "z", false )); // [ballroom, dancing]

    out.println("\nClosest-matches:");               // (4)
    out.println(strSetA.ceiling("ball"));            // ballroom
    out.println(strSetA.floor("ball"));              // Strictly
    out.println(strSetA.higher("ballroom"));         // dancing
    out.println(strSetA.lower("ballroom"));          // Strictly
    out.println("\nReverse order:");                    // (5)
    out.println(strSetA.descendingSet());   // [dancing, ballroom, Strictly, Java]

    out.println("\nFirst-last elements:");
    out.println(strSetA.pollFirst());                   // Java
    out.println(strSetA.pollLast());                    // dancing

    out.println("\nAfter: " + strSetA);                 // [Strictly, ballroom]
  }
}
```

```
Before: [Java, Strictly, ballroom, dancing]

Subset-views:
[Java, Strictly, ballroom]
[Java, Strictly]
[Strictly, ballroom, dancing]
[ballroom, dancing]
[Java, Strictly]
[ballroom, dancing]

Closest-matches:
ballroom
Strictly
dancing
Strictly

Reverse order:
[dancing, ballroom, Strictly, Java]

First-last elements:
Java
dancing

After: [Strictly, ballroom]
```

```
PriorityQueue()
PriorityQueue(int initialCapacity)
PriorityQueue(Comparator<? super E> comparator)
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
PriorityQueue(PriorityQueue<? extends E> pq)
PriorityQueue(SortedSet<? extends E> set)
PriorityQueue(Collection<? extends E> c)
```

```
private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {    // (8)
...
}
```

```
PriorityQueue<Task> pq1 = new PriorityQueue<>();                           // (2)
```

```
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
```

Task at the head: 100@breakfast

```
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner
```

```
PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
```

```
PriorityQueue<Task> pq4 = new PriorityQueue<>(                              // (5)
    Comparator.comparing(Task::getTaskName).reversed()
);
```

```
PriorityQueue<Task> pq6 = new PriorityQueue<>(                           // (7)
      Comparator.comparing(Task::getTaskNumber)
                .thenComparing(Task::getTaskName)
);
```

```
/** Represents a task. */
public class Task implements Comparable<Task> {
  private Integer taskNumber;
  private String  taskName;

  public Task(Integer tp, String tn) {
    taskNumber = tp;
    taskName   = tn;
  }

  @Override
  public boolean equals(Object obj) {// Equality based on the task number.
    return (this == obj)
        || (obj instanceof Task other
            && this.taskNumber.equals(other.taskNumber));
  }
  @Override
  public int compareTo(Task task2) { // Natural ordering based on the task number.
    return this.taskNumber.compareTo(task2.taskNumber);
  }
  @Override
  public int hashCode() {            // Hash code based on the task number.
    return this.taskNumber.hashCode();
  }
  @Override
  public String toString() { return taskNumber + "@" + taskName; }

  public String  getTaskName()   { return taskName; }
  public Integer getTaskNumber() { return taskNumber;}
}
```

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;
import static java.lang.System.out;
```

```
/** Executes tasks. */
public class TaskExecutor {

  public static void main(String[] args) {
    // Array with some tasks.                                                  (1)
    Task[] taskArray = {
        new Task(200, "lunch"), new Task(200, "tea"),
        new Task(300, "dinner"), new Task(100, "breakfast"),
    };
    out.println("Array of tasks: " + Arrays.toString(taskArray));

    out.println("Priority queue using natural ordering (task number).");
    PriorityQueue<Task> pq1 = new PriorityQueue<>();                          // (2)
    testPQ(taskArray, pq1);

    out.println("Priority queue using reverse natural ordering.");           // (3)
    PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
    testPQ(taskArray, pq2);

    out.println("Priority queue using reversed ordering"
        + " on task name (lambda expression).");
    PriorityQueue<Task> pq3 = new PriorityQueue<>(                           // (4)
        (task1, task2) -> {
          String taskName1 = task1.getTaskName();
          String taskName2 = task2.getTaskName();
          return -taskName1.compareTo(taskName2);
         }
      );
    testPQ(taskArray, pq3);

    out.println("Priority queue using reversed ordering"
        + " on task name (extracted comparator).");
    PriorityQueue<Task> pq4 = new PriorityQueue<>(                           // (5)
        Comparator.comparing(Task::getTaskName).reversed()
    );
    testPQ(taskArray, pq4);

    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (lambda expression).");
    PriorityQueue<Task> pq5 = new PriorityQueue<>(                           // (6)
      (task1, task2) -> {
        Integer taskNumber1 = task1.getTaskNumber();
        Integer taskNumber2 = task2.getTaskNumber();
        if (!taskNumber1.equals(taskNumber2))
          return taskNumber1.compareTo(taskNumber2);
        String taskName1 = task1.getTaskName();
        String taskName2 = task2.getTaskName();
        if (!taskName1.equals(taskName2))
          return taskName1.compareTo(taskName2);
        return 0;
       }
    );
    testPQ(taskArray, pq5);
    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (extracted comparators).");
    PriorityQueue<Task> pq6 = new PriorityQueue<>(                           // (7)
        Comparator.comparing(Task::getTaskNumber)
                  .thenComparing(Task::getTaskName)
    );
    testPQ(taskArray, pq6);
  }

  // Runs tasks.
  private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {    // (8)
    // Load the tasks:                                                          (9)
    for (Task task : taskArray)
      pq.offer(task);
    out.println("Queue before executing tasks: " + pq);

    // Peek at the head:                                                       (10)
    out.println("Task at the head: " + pq.peek());

    // Do the tasks:                                                           (11)
    out.print("Doing tasks: ");
    while (!pq.isEmpty()) {
      Task task = pq.poll();
      out.print(task + " ");
    }
    out.println();
    out.println();
  }
}
```

```
Array of tasks: [200@lunch, 200@tea, 300@dinner, 100@breakfast]
Priority queue using natural ordering (task number).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner

Priority queue using reverse natural ordering.
Queue before executing tasks: [300@dinner, 200@tea, 200@lunch, 100@breakfast]
Task at the head: 300@dinner
Doing tasks: 300@dinner 200@tea 200@lunch 100@breakfast

Priority queue using reversed ordering on task name (lambda expression).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using reversed ordering on task name (extracted comparator).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using total ordering based on task number,
followed by task name (lambda expression).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
```

```
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner

Priority queue using total ordering based on task number,
followed by task name (extracted comparators).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner
```

```
// Insert
boolean offerFirst(E element)
boolean offerLast(E element)                 Queue equivalent: offer()
void addFirst(E element)
void addLast(E element)                      Queue equivalent:  add()
void push(E element)                         Synonym: addFirst()
```

```
// Remove
E removeFirst()                               Queue equivalent: remove()
E removeLast()
E pollFirst()                                 Queue equivalent: poll()
E pollLast()
E pop()                                       Synonym: removeFirst()
boolean removeFirstOccurence(Object obj)
boolean removeLastOccurence(Object obj)
```

```
// Examine
E getFirst()                                   Queue equivalent: element()
E getLast()
E peekFirst()                                  Queue equivalent: peek()
E peekLast()
```

```
// Misc.
Iterator<E> descendingIterator()
```

```
ArrayDeque(Collection<? extends E> c)
```

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeOperations {
  public static void main(String[] args) {
    Deque<String> deque = new ArrayDeque<String>();
    System.out.println("After creating the deque: " + deque);

    // Insert elements:
    deque.offerFirst("A (H)");               // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.offerLast("B (T)");                // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);
    deque.push("C (H)");                     // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addFirst("D (H)");                 // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addLast("E (T)");                  // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);

    // Examine element:
    System.out.println("Examine at the head: " + deque.getFirst());
    System.out.println("Examine at the tail: " + deque.getLast());
    System.out.println("Examine at the head: " + deque.peekFirst());
    System.out.println("Examine at the tail: " + deque.peekLast());

    // Remove elements:
    deque.removeFirst();                     // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.removeLast();                      // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pollFirst();                       // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.pollLast();                        // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pop();                             // Remove from the head
    System.out.println("After removing from the head: " + deque);
  }
}
```

```
After creating the deque: []
After inserting at the head: [A (H)]
After inserting at the tail: [A (H), B (T)]
After inserting at the head: [C (H), A (H), B (T)]
After inserting at the head: [D (H), C (H), A (H), B (T)]
After inserting at the tail: [D (H), C (H), A (H), B (T), E (T)]
Examine at the head: D (H)
Examine at the tail: E (T)
Examine at the head: D (H)
Examine at the tail: E (T)
After removing from the head: [C (H), A (H), B (T), E (T)]
After removing from the tail: [C (H), A (H), B (T)]
After removing from the head: [A (H), B (T)]
After removing from the tail: [A (H)]
After removing from the head: []
```

```
import java.util.ArrayDeque;
import java.util.Arrays;

/** Executes tasks. */
public class TaskExecutor2 {

  public static void main(String[] args) {
    String[] elementArray = {"sway", "and", "twist", "stacks", "tall"};     // (1)
    System.out.println("Array of elements: " + Arrays.toString(elementArray));

    // Using ArrayDeque as a stack:                                            (2)
    ArrayDeque<String> stack = new ArrayDeque<>();
    for (String string : elementArray)
      stack.push(string);                             // (3) Push elements.
    System.out.println("Stack before: TOP->" + stack + "<-BOTTOM");
    System.out.print("Popping stack: ");
    while (!stack.isEmpty()) {                        // (4)
      System.out.print(stack.pop() + " ");            // (5) Pop elements.
    }
    System.out.println("\n");

    // Using ArrayDeque as a FIFO queue:                 (6)
    elementArray = new String[] {"Waiting", "in", "queues", "is", "boring"};// (7)
    System.out.println("Array of elements: " + Arrays.toString(elementArray));
    ArrayDeque<String> fifoQueue = new ArrayDeque<>();
    for (String string : elementArray)
      fifoQueue.offerLast(string);                    // (8) Insert at tail.
    System.out.println("Queue before: HEAD->" + fifoQueue  + "<-TAIL");
    System.out.print("Polling queue: ");
    while (!fifoQueue.isEmpty()) {                    // (9)
      String string = fifoQueue.pollFirst();          // (10) Remove from head.
      System.out.print(string.toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

```
Array of elements: [sway, and, twist, stacks, tall]
Stack before: TOP->[tall, stacks, twist, and, sway]<-BOTTOM
Popping stack: tall stacks twist and sway

Array of elements: [Waiting, in, queues, is, boring]
Queue before: HEAD->[Waiting, in, queues, is, boring]<-TAIL
Polling queue: WAITING IN QUEUES IS BORING
```

```
import java.util.*;
public class RQ400A100 {
  public static void main(String[] args) {
    int sum = 0;
    for (int i : makeCollection()) {
      sum += i;
    }
    System.out.println(sum);
  }

  static Collection<Integer> makeCollection() {
    System.out.println("A collection coming up.");
    Collection<Integer> collection = new ArrayList<>();
    collection.add(10); collection.add(20); collection.add(30);
    return collection;
  }
}
```

```
import java.util.*;
class Fruity {
  private String fName;
  Fruity(String fName) { this.fName = fName; }
  public void setName(String newName) { this.fName = newName; }
  public String toString() { return fName; }
  public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof Fruity)) return false;
    return fName.equalsIgnoreCase(((Fruity)other).fName);
  }
}

public class RQ400A50 {
  public static void main(String[] args) {
    Fruity apple = new Fruity("Apple");
    Fruity orange = new Fruity("Orange");
    List<Fruity> list = new ArrayList<>();
    list.add(apple); list.add(orange); list.add(apple);
    System.out.println("Before: " + list);

    // (1) INSERT CODE HERE

    System.out.println("After:  " + list);
  }
}
```

```
Before: [Apple, Orange, Apple]
After:  [Orange]
```

```
for (int j = 0; j < list.size(); j++) {
  Fruity f = list.get(j);
  if (f.equals(apple))
    list.remove(j);
}
```

```
Iterator<Fruity> itr = list.iterator();
while (itr.hasNext()) {
  Fruity f = itr.next();
  if (f.equals(apple))
    itr.remove();
}
```

```
import java.util.*;
class AnotherListIterator<T> implements Iterable<T>{

  private List<T> lst;
  public AnotherListIterator(List<T> lst) { this.lst = lst; }

  public Iterator<T> iterator() {
    return new Iterator<T>() {
      private int next = lst.size() - 1;

      public boolean hasNext() { return (next >= 0); }
      public T next() {
        T element = lst.get(next);
        next--;
        return element;
      }
    };
  }

  public static void main(String[] args) {
    List<String> lst = List.of("Hi", "Howdy", "Hello");
    AnotherListIterator<String> rlt = new AnotherListIterator<>(lst);
    for (String str : rlt) {
      System.out.print("|" + str + "|");
    }
  }
}
```

```
import java.util.*;
public class Sets {
  public static void main(String[] args) {
    HashSet<Integer> set1 = new HashSet<>();
    addRange(set1, 1);
    ArrayList<Integer> list1 = new ArrayList<>();
    addRange(list1, 2);
    TreeSet<Integer> set2 = new TreeSet<>();
    addRange(set2, 3);
    ArrayDeque<Integer> deque = new ArrayDeque<>();
    addRange(deque, 5);
    set1.removeAll(list1);
    list1.addAll(set2);
    deque.addAll(list1);
    set1.removeAll(deque);
    System.out.println(set1);
  }
  static void addRange(Collection<Integer> col, int step) {
    for (int i = step * 2; i <= 25; i += step) {
      col.add(i);
    }
  }
}
```

```
import java.util.*;
public class Iterate {
  public static void main(String[] args) {
    List<String> l = new ArrayList<>();
    l.add("A"); l.add("B"); l.add("C"); l.add("D"); l.add("E");
    ListIterator<String> i = l.listIterator();
    i.next(); i.next(); i.next(); i.next();
    i.remove();
    i.previous(); i.previous();
    i.remove();
    System.out.println(l);
  }
}
```

```
import java.util.*;
public class RightOne {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    collection.add(1); collection.add(9); collection.add(1);
    System.out.println(collection);
  }
}
```

```
Collection<Integer> collection = new HashSet<>();
```

```
Set<Integer> collection = new HashSet<>();
```

```
HashSet<Integer> collection = new LinkedHashSet<>();
```

```
Set<Integer> collection = new LinkedHashSet<>();
```

```
Collection<Integer> collection = new TreeSet<>();
```

```
NavigableSet<Integer> collection = new TreeSet<>();
```

```
import static java.lang.System.out;
import java.util.*;
public class RQ400A400 {
  public static void main(String[] args) {
    NavigableSet<String> strSetA = new TreeSet<>();
    Collections.addAll(strSetA, "sea", "shell", "soap", "swan");
    // (1) INSERT STATEMENT HERE
  }
}
```

```
out.println(strSetA.headSet("soap", true));
out.println(strSetA.headSet("soap", false));
out.println(strSetA.tailSet("soap", true));
out.println(strSetA.tailSet("soap", false));
out.println(strSetA.subSet("sea", false, "soap", true));
out.println(strSetA.subSet("sea", true, "soap", false));
```

`V put(K key, V value)`	Optional
`default V putIfAbsent(K key, V value)`	

```
V get(Object key)
default V getOrDefault(Object key, V defaultValue)
```

`V remove(Object key)`	Optional
`default boolean remove(Object key, Object value)`	

```
default V replace(K key, V value)
default boolean replace(K key, V oldValue, V newValue)
```

```
boolean containsKey(Object key)
boolean containsValue(Object value)
```

```
static <K,?V> Map<K,?V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
                           K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
                           K k9, V v9, K k10, V v10)
```

```
@SafeVarargs static <K,V> Map<K,V> ofEntries(
             Map.Entry<? extends K,? extends V>... entries)
```

```
static <K,V> Map.Entry<K,V> entry(K k, V v)
```

```
interface Entry<K, V> {    // Nested interface in the Map<K, V> interface.
    K getKey();
    V getValue();
    V setValue(V value);   // Only if the entry is modifiable.
}
```

```
static <K,V> Map<K,V> copyOf(Map<? extends K, ? extends V> map)
```

```
Map<Integer, String> jCourses = Map.of(
                     200, "Basic Java",    300, "Intermediate Java",
                     400, "Advanced Java", 500, "Kickass Java");
//  jCourses.put(200, "Java Jive");               // UnsupportedOperationException
//  jCourses.remove(500);                         // UnsupportedOperationException
System.out.println(jCourses instanceof HashMap); // false
System.out.println(jCourses);
// {200=Basic Java, 400=Advanced Java, 300=Intermediate Java, 500=Kickass Java}
```

```
Map<Integer, String> coursesMap1
        = Map.of(101, "Java 1.1", 101, "Java 17");  // IllegalArgumentException
Map<Integer, String> coursesMap2
        = Map.of(101, "Java 1.1", 101, null);       // NullPointerException
```

```
//  Map.Entry<Integer, String> e0 = Map.entry(100, null); // NullPointerException
Map.Entry<Integer, String> e1 = Map.entry(200, "Basic Java");
Map.Entry<Integer, String> e2 = Map.entry(300, "Intermediate Java");
Map.Entry<Integer, String> e3 = Map.entry(400, "Advanced Java");
Map.Entry<Integer, String> e4 = Map.entry(500, "Kickass Java");
```

```
Map<Integer, String> unmodCourseMap = Map.ofEntries(e1, e2, e3, e4);  // Varargs
// {300=Intermediate Java, 500=Kickass Java, 200=Basic Java, 400=Advanced Java}
//unmodCourseMap.replace(200, "Java Jive");    // UnsupportedOperationException
//unmodCourseMap.remove(500);                  // UnsupportedOperationException
```

```
StringBuilder mc1 = new StringBuilder("Basic Java");
StringBuilder mc2 = new StringBuilder("Intermediate Java");
StringBuilder mc3 = new StringBuilder("Advanced Java");
StringBuilder mc4 = new StringBuilder("Kickass Java");
```

```
Map.Entry<Integer, StringBuilder> me1 = Map.entry(200, mc1);
Map.Entry<Integer, StringBuilder> me2 = Map.entry(300, mc2);
Map.Entry<Integer, StringBuilder> me3 = Map.entry(400, mc3);
Map.Entry<Integer, StringBuilder> me4 = Map.entry(500, mc4);
```

```
Map<Integer, StringBuilder> unmodMapWithMutableCourses
             = Map.ofEntries(me1, me2, me3, me4);         // Varargs
System.out.println(unmodMapWithMutableCourses);
// {200=Basic Java, 500=Kickass Java, 300=Intermediate Java, 400=Advanced Java}

// unmodMapWithMutableCourses.replace(200, mc4); // UnsupportedOperationException
// unmodMapWithMutableCourses.remove(400);       // UnsupportedOperationException
```

```
StringBuilder mutableCourse = unmodMapWithMutableCourses.get(500);
mutableCourse.replace(0, 7, "Smartass");
```

```
System.out.println(unmodMapWithMutableCourses);
// {400=Advanced Java, 500=Smartass Java, 200=Basic Java, 300=Intermediate Java}
// Original map:
Map<Integer, StringBuilder> courseMap = new HashMap<>();
courseMap.put(200, mc1); courseMap.put(300, mc2);
courseMap.put(400, mc3); courseMap.put(500, mc4);

// Unmodifiable copy of the map:
Map<Integer, StringBuilder> copyCourseMap = Map.copyOf(courseMap); // (1)

// Modify original map:
courseMap.remove(200);
courseMap.remove(400);
System.out.println("Original: " + courseMap);
System.out.println("Copy: " + copyCourseMap);
```

```
Original: {500=Smartass Java, 300=Intermediate Java}
Copy: {300=Intermediate Java, 500=Smartass Java, 200=Basic Java,
       400=Advanced Java}
```

```
default V merge(K key, V value,
                BiFunction<? super V,? super V,? extends V> remappingFunc)
```

```
default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remappingFunc)
```

```
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunc)
```

```
default V computeIfPresent(K key,
                 BiFunction<? super K,? super V,? extends V> remappingFunc)
```

```
Map<Integer, String> etnMap = new HashMap<>();
etnMap.put(112, "Norway");
etnMap.put(999, "UK");
etnMap.put(190, null);
etnMap.put(911, null);
// {112=Norway, 999=UK, 190=null, 911=null}
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.merge(112, "Mordor", (oVal, value) -> null);      // (1) null, removed
etnMap.merge(999, "Mordor", (oVal, value) -> "Uganda");  // (2) Uganda, updated
etnMap.merge(190, "Mordor", (oVal, value) -> null);      // (3) Mordor, updated
etnMap.merge(911, "Mordor", (oVal, value) -> "USA");     // (4) Mordor, updated
etnMap.merge(100, "Mordor", (oVal, value) -> null);      // (5) Mordor, inserted
etnMap.merge(110, "Mordor", (oVal, value) -> "China");   // (6) Mordor, inserted
// After: {100=Mordor, 999=Uganda, 110=Mordor, 190=Mordor, 911=Mordor}
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.compute(112, (key, oVal) -> null);                // (7) null, removed
etnMap.compute(999, (key, oVal) -> "Uganda");            // (8) Uganda, updated
etnMap.compute(190, (key, oVal) -> null);                // (9) null, removed
etnMap.compute(911, (key, oVal) -> "USA");               // (10) USA, updated
etnMap.compute(100, (key, oVal) -> null);                // (11) null, no action
etnMap.compute(110, (key, oVal) -> "China");             // (12) China, inserted
// After: {110=China, 999=Uganda, 911=USA}
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfAbsent(112, key -> null);             // (13) Norway, no change
etnMap.computeIfAbsent(999, key -> "Uganda");         // (14) UK, no change
etnMap.computeIfAbsent(190, key -> null);             // (15) null, no change
etnMap.computeIfAbsent(911, key -> "USA");            // (16) USA, updated
etnMap.computeIfAbsent(100, key -> null);             // (17) null, no action
etnMap.computeIfAbsent(110, key -> "China");          // (18) China, inserted
// After: {112=Norway, 110=China, 999=UK, 190=null, 911=USA}
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfPresent(112, (key, oVal) -> null);      // (19) null, removed
etnMap.computeIfPresent(999, (key, oVal) -> "Uganda");  // (20) Uganda, updated
etnMap.computeIfPresent(190, (key, oVal) -> null);      // (21) null, no change
etnMap.computeIfPresent(911, (key, oVal) -> "USA");     // (22) null, no change
etnMap.computeIfPresent(100, (key, oVal) -> null);      // (23) null, no action
etnMap.computeIfPresent(110, (key, oVal) -> "China");   // (24) null, no action
// After: {999=Uganda, 190=null, 911=null}
```

`void clear()`	Optional
`void putAll(Map<? extends K, ? extends V> map)`	Optional

```
default void forEach(BiConsumer<? super K,? super V> action)
default void replaceAll(BiFunction<? super K,? super V,? extends V> func)
```

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
HashMap(Map<? extends K,? extends V> otherMap)
```

```
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```

```
// With method reference:
groupFreqMap.merge(weightGroup, 1, Integer::sum);                    // (5)
// With lambda expression:
groupFreqMap.merge(weightGroup, 1,
   (oldVal, givenVal) -> Integer.sum(oldVal, givenVal));
```

```
Integer frequency = groupFreqMap.get(weightGroup);
if (frequency == null) frequency = 0;
groupFreqMap.put(weightGroup, ++frequency);
```

```
Integer frequency = groupFreqMap.getOrDefault(weightGroup, 0);
groupFreqMap.put(weightGroup, ++frequency);
groupFreqMap.putIfAbsent(weightGroup, 0);
Integer frequency = groupFreqMap.get(weightGroup);
groupFreqMap.put(weightGroup, ++frequency);
groupFreqMap.compute(weightGroup, (k, v) -> v == null ? 1 : v + 1);
```

```
TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet()); // (6)
```

```
sortedKeySet.forEach(key ->                                         // (7)
    System.out.printf("%5s: %s%n", key,
        String.join("", Collections.nCopies(groupFreqMap.get(key), "*"))))
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeSet;

public class Histogram {
  public static void main(String[] args) {
    System.out.println("Data: " + Arrays.toString(args));

    // Create a map to store the frequency for each group.
    Map<Integer, Integer> groupFreqMap = new HashMap<>();                  // (1)

    // Determine the frequencies:
    for (String argument : args) {                                         // (2)
      double weight = Double.parseDouble(argument);                        // (3)
      int weightGroup = (int) Math.round(weight/5.0)*5;                    // (4)
      groupFreqMap.merge(weightGroup, 1, Integer::sum);                    // (5)
    }
    System.out.println("Frequencies: " + groupFreqMap);

    // Create sorted key set.
    TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet());  // (6)

    System.out.println("Histogram:");
    // Implicit iteration over the key set.
    sortedKeySet.forEach(key ->                                            // (7)
        System.out.printf("%5s: %s%n", key,
            String.join("", Collections.nCopies(groupFreqMap.get(key), "*")))
    );
  }
}
```

```
>java Histogram 74 75 93 75 93 82 61 92 10 185
```

```
Data: [74, 75, 93, 75, 93, 82, 61, 92, 10, 185]
Frequencies: {80=1, 185=1, 10=1, 90=1, 75=3, 60=1, 95=2}
Histogram:
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
```

```
// First-last keys
K firstKey()                                         Sorted set: first()
K lastKey()                                          Sorted set: last()
```

```
// Range-view operations
SortedMap<K,V> headMap(K toKey)                    Sorted set: headSet()
SortedMap<K,V> tailMap(K fromKey)                  Sorted set: tailSet()
SortedMap<K,V> subMap(K fromKey, K toKey)          Sorted set: subSet()
```

```
// Comparator access
Comparator<? super K> comparator()
```

```
// First-last entries
// Remove
Map.Entry<K, V> pollFirstEntry()                    Navigable set: pollFirst()
Map.Entry<K, V> pollLastEntry()                     Navigable set: pollLast()

// Examine
Map.Entry<K, V> firstEntry()
Map.Entry<K, V> lastEntry()
```

```
// Range-view operations
NavigableMap<K, V> headMap(K toKey,                  Navigable set: headSet()
                     boolean inclusive)
NavigableMap<K, V> tailMap(K fromKey,                Navigable set: tailSet()
                     boolean inclusive)
NavigableMap<K, V> subMap(K fromKey,                 Navigable set: subSet()
                     boolean fromInclusive,
                     K toKey,
                     boolean toInclusive)
```

```
// Closest-matches
Map.Entry<K, V> ceilingEntry(K key)                    Navigable set: ceiling()
K               ceilingKey(K key)
Map.Entry<K, V> floorEntry(K key)                      Navigable set: floor()
K               floorKey(K key)
Map.Entry<K, V> higherEntry(K key)                     Navigable set: higher()
K               higherKey(K key)
Map.Entry<K, V> lowerEntry(K key)                      Navigable set: lower()
K               lowerKey(K key)
```

```
// Navigation-views
NavigableMap<K, V> descendingMap()
NavigableSet<K> descendingKeySet()
NavigableSet<K> navigableKeySet()
```

Navigable set: `descendingSet()`

```
TreeMap(Comparator<? super K> c)
```

`TreeMap(Map<? extends K, ? extends V> map)`

```
TreeMap(SortedMap<K, ? extends V> map)
```

```
public static <K> void printHistogram(NavigableMap<K, Integer> freqMap) {...}
```

```
Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
No. of weight groups: 7
...
```

```
out.println("First entry: " + groupFreqMap.firstEntry());                // (4)
out.println("Last entry: " + groupFreqMap.lastEntry());                  // (5)
out.println("Greatest entry <= 77: "
           + groupFreqMap.floorEntry(77));                          // (6)
out.println("Smallest key > 90: "
           + groupFreqMap.higherKey(90));                           // (7)
```

```
Greatest entry <= 77: 75=3
Smallest key > 90: 95
```

```
...
printHistogram(groupFreqMap.tailMap(75, true));                          // (8)
...
printHistogram(groupFreqMap.headMap(75, false));                         // (9)
```

```
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
...
Head map (Groups <  75): {10=1, 60=1}
...
```

```
printHistogram(groupFreqMap.subMap(                                  // (10)
     groupFreqMap.firstEntry().getKey(), false,
     groupFreqMap.lastEntry().getKey(), false));
```

```
Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1, 95=2}
...
```

```
// Poll the navigable map:                                          // (11)
int sumValues = 0;
while (!groupFreqMap.isEmpty()) {
  Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();   // (12)
  Integer frequency = entry.getValue();
  sumValues += frequency;
  out.printf("%5s: %s%n", entry.getKey(), frequency);
}
```

```
import java.util.Collections;
import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;
import static java.lang.System.out;

public class HistogramStats {
  public static void main(String[] args) {

    // Create a navigable map to store the frequency for each group.
    NavigableMap<Integer, Integer> groupFreqMap = new TreeMap<>();         // (1)

    // Determine the frequencies:
    for (String argument : args) {                                         // (2)
      double weight = Double.parseDouble(argument);
      int weightGroup = (int) Math.round(weight/5.0)*5;
      groupFreqMap.merge(weightGroup, 1, Integer::sum);
    }

    // Print statistics about the frequency map:
    out.print("Group frequency map: ");
    printHistogram(groupFreqMap);                                          // (3)

    out.println("First entry: " + groupFreqMap.firstEntry());              // (4)
    out.println("Last entry: " + groupFreqMap.lastEntry());                // (5)
    out.println("Greatest entry <= 77: "
                + groupFreqMap.floorEntry(77));                            // (6)
    out.println("Smallest key > 90: "
                + groupFreqMap.higherKey(90));                             // (7)

    out.print("Tail map (Groups >= 75): ");
    printHistogram(groupFreqMap.tailMap(75, true));                        // (8)

    out.print("Head map (Groups <  75): ");
    printHistogram(groupFreqMap.headMap(75, false));                       // (9)

    out.print("Frequency map (first and last entry excluded): ");
    printHistogram(groupFreqMap.subMap(                                    // (10)
        groupFreqMap.firstEntry().getKey(), false,
        groupFreqMap.lastEntry().getKey(), false));

    // Poll the navigable map:                                                (11)
    out.println("Histogram (by polling):");
    int sumValues = 0;
    while (!groupFreqMap.isEmpty()) {
      Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();   // (12)
      Integer frequency = entry.getValue();
      sumValues += frequency;
      out.printf("%5s: %s%n", entry.getKey(), frequency);
    }
    out.println("Number of weights registered: " + sumValues);             // (13)
    out.println("Group frequency map after polling: " + groupFreqMap);     // (14)
  }

  // Prints a histogram for entries in a navigable map.
  public static <K> void printHistogram(NavigableMap<K, Integer> freqMap) {// (15)
    out.println(freqMap);                                                  // (16)
    out.println("No. of entries: " + freqMap.size());                      // (17)
    freqMap.forEach((k, v) ->                                              // (18)
        out.printf("%5s: %s%n", k,
                   String.join("", Collections.nCopies(v, "*"))));
  }
}
```

```
>java HistogramStats 74 75 93 75 93 82 61 92 10 185
```

```
Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
No. of entries: 7
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
First entry: 10=1
Last entry: 185=1
```

```
Greatest entry <= 77: 75=3
Smallest key > 90: 95
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
No. of entries: 5
   75: ***
   80: *
   90: *
   95: **
  185: *
Head map (Groups <  75): {10=1, 60=1}
No. of entries: 2
   10: *
   60: *
Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1, 95=2}
No. of entries: 5
   60: *
   75: ***
   80: *
   90: *
   95: **
Histogram (by polling):
   10: 1
   60: 1
   75: 3
   80: 1
   90: 1
   95: 2
  185: 1
Number of weights registered: 10
Group frequency map after polling: {}
```

```
import java.util.*;
public class TripleJump1 {
  public static void main(String[] args) {
    NavigableSet<String> set = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set, "Step", "Jump", "Step", "Hop");
    System.out.println(set);
  }
}
```

```
import java.util.*;
public class TripleJump2 {
  public static void main(String[] args) {
    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>(set1);
    System.out.println(set2);
  }
}
```

```
import java.util.*;
public class TripleJump3 {
  public static void main(String[] args) {
    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
```

```
        System.out.println(set2);
    }
}
```

```
import java.util.*;
public class TripleJump4 {
    public static void main(String[] args) {
    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
    while (!set1.isEmpty()) {
      System.out.print(set1.pollLast() + " ");
    }
    while (!set2.isEmpty()) {
      System.out.print(set2.pollFirst() + " ");
    }
  }
}
```

```
import java.util.*;
public class MapModify {
  public static void main(String[] args) {
    NavigableMap<String, Integer> grades = new TreeMap<>();
    grades.put("A",  5); grades.put("B", 10); grades.put("C", 15);
    grades.put("D", 20); grades.put("E", 25);

    System.out.print(grades.get(grades.firstKey()) + " ");
    System.out.print(sumValues(grades.headMap("D")) + " ");
    System.out.print(sumValues(grades.subMap("B", false, "D", true)) + " ");
    grades.subMap(grades.firstKey(), false, grades.lastKey(), false).clear();
    System.out.println(sumValues(grades));
  }
  public static <K, M extends Map<K, Integer>> int sumValues(M freqMap) {
    return freqMap.values().stream().mapToInt(i->i).sum();
  }
}
```

```
import java.util.*;
public class Mapping {
  public static void main(String[] args) {
    NavigableMap<String, Integer> myMap
                   = new TreeMap<>(Collections.reverseOrder());
    myMap.put("Soap", 10); myMap.put("Shampoo", 20); myMap.put("Salts", 30);
    // (1) INSERT CODE HERE
    System.out.println(myMap);
  }
}
```

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
    myMap.remove("Shampoo");
```

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();)
  if (iterator.next().equals("Shampoo"))
    iterator.remove();
```

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();) {
  if (iterator.next().equals("Shampoo"))
    myMap.remove("Shampoo");
```

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
    myMap.remove(entry);
myMap.subMap("Shampoo", true, "Shampoo", true).clear();
myMap.compute("Shampoo", (k, v) -> null);
```

```
import java.util.*;
public class Valuables {
  public static void main(String[] args) {
    NavigableMap<Integer, Integer> iMap = new TreeMap<>();
    iMap.put(100, 1); iMap.put(200, 2); iMap.put(300, 3);
    int sumVal = 0;
    iMap.forEach((k, v) -> sumVal += v);
    System.out.println(sumVal);
  }
}
```

```
import java.util.*;
public class InitMap {
  public static void main(String[] args) {
    Map<Integer, NavigableSet<String>> etnMap = new TreeMap<>();
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("USA");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Canada");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Argentina");
    etnMap.computeIfAbsent(112, key -> new TreeSet<>()).add("Norway");
    System.out.println(etnMap);
  }
}
```

```
import java.util.*;
import java.util.function.*;
public class Test13RQ13 {
  public static void main(String[] args) {
    Map<Integer, String> values = new HashMap<>();
    values.put(1, "ONE");   values.put(2, "TWO");
    values.put(3, "THREE"); values.put(4, "FOUR");
    values.replaceAll((k, v) -> {
      switch (k) {
        case 1: return "FIRST";
        case 2: return "SECOND";
        case 3: return "THIRD";
        case 4: return "FOURTH";
      }
      return "ZERO";
    });
    values.forEach((Integer x, String y) -> { System.out.print(y + " "); } );
  }
}
```

```
<E> Collection<E>   unmodifiableCollection(Collection<? extends E> c)
<E> List<E>         unmodifiableList(List<? extends E> list)
<E> Set<E>          unmodifiableSet(Set<? extends E> set)
<E> SortedSet<E>    unmodifiableSortedSet(SortedSet<E> sortedSet)
<E> NavigableSet<E> unmodifiableNavigableSet(NavigableSet<E> navSet)
```

```
<K, V> Map<K, V>        unmodifiableMap(Map<? extends K, ? extends V> map)
<K, V> SortedMap<K, V>  unmodifiableSortedMap(SortedMap<K, ? extends V> sm)
<K, V> NavigableMap<K, V>
                  unmodifiableNavigableMap(NavigableMap<K, ? extends V> nm)
```

```
// Mutable courses:
StringBuilder mc1 = new StringBuilder("Java I");
StringBuilder mc2 = new StringBuilder("Java II");
StringBuilder mc3 = new StringBuilder("Java III");
StringBuilder mc4 = new StringBuilder("Java IV");

// Backing map:
Map<Integer, StringBuilder> backingMap = new HashMap<>();
backingMap.put(200, mc1); backingMap.put(300, mc2);
backingMap.put(400, mc3); backingMap.put(500, mc4);
```

```
// Unmodifiable view of a map:
Map<Integer, StringBuilder> unmodViewMap
               = Collections.unmodifiableMap(backingMap);
```

```
 // UnsupportedOperationException at (1), (2), and (3):
unmodViewMap.put(100, new StringBuilder("Java Now"));
unmodViewMap.remove(400);
unmodViewMap.replace(200, new StringBuilder("First Java"));
```

```
backingMap.remove(200); backingMap.remove(400);
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {500=Java Complete, 300=Java II}
//  Unmodifiable view: {500=Java Complete, 300=Java II}
```

```
StringBuilder mutableCourse = unmodViewMap.get(500);  // Get the course.
mutableCourse.replace(5, 8, "Complete");              // Change the course name.
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {400=Java III, 500=Java Complete, 200=Java I, 300=Java II}
//  Unmodifiable view: {400=Java III, 500=Java Complete, 200=Java I, 300=Java II}
```

```
<E extends Comparable<? super E>> void sort(List<E> list)
<E> void sort(List<E> list, Comparator<? super E> comparator)
```

```
<E> Comparator<E> reverseOrder()
<E> Comparator<E> reverseOrder(Comparator<E> comparator)
```

```
// Defined in the List <E> interface.
void sort(Comparator<? super E> comparator)
```

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "biggest", "big", "bigger", "Bigfoot");

strList.sort(null);                                        // Natural order
strList.sort(Comparator.comparing(String::length));      // length order
strList.sort(Collections.reverseOrder());                // Reverse natural order
Collections.sort(strList, String.CASE_INSENSITIVE_ORDER);// Case insensitive order
Collections.sort(strList,                        // Reverse case insensitive order
                 Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

```
Before sorting:                            [biggest, big, bigger, Bigfoot]
After sorting in natural order:            [Bigfoot, big, bigger, biggest]
After sorting in length order:             [big, bigger, Bigfoot, biggest]
After sorting in reverse natural order:    [biggest, bigger, big, Bigfoot]
After sorting in insensitive order:        [big, Bigfoot, bigger, biggest]
After sorting in reverse insensitive order: [biggest, bigger, Bigfoot, big]
```

```
List<StringBuilder> sbList = new ArrayList<>();
Collections.addAll(sbList, new StringBuilder("smallest"),
                   new StringBuilder("small"), new StringBuilder("smaller"));
Collections.sort(sbList);                 // [small, smaller, smallest]
```

```
List freakList = new ArrayList();                    // Raw types.
Collections.addAll(freakList, 23, 3.14, 10L);
freakList.sort(null);                                // ClassCastException
```

```
Set<Integer> intSet = new TreeSet<>(Collections.reverseOrder());
Collections.addAll(intSet, 9, 11, -4, 1);
System.out.println(intSet);          // [11, 9, 1, -4]
```

```
void rotate(List<?> list, int distance)
void shuffle(List<?> list)
void shuffle(List<?> list, Random rnd)
void swap(List<?> list, int i, int j)
```

```
// intList refers to the following list:                    [9, 11, -4, 1, 7]
Collections.rotate(intList, 2);         // Two to the right.     [1, 7, 9, 11, -4]
Collections.rotate(intList, -2);        // Two to the left.      [9, 11, -4, 1, 7]
List intSublist = intList.subList(1,4);// Sublist:                  [11, -4, 1]
Collections.rotate(intSublist, -1);     // One to the left.          [-4, 1, 11]
                                        // intList is now:       [9, -4, 1, 11, 7]
```

```
<E> int binarySearch(List<? extends Comparable<? super E>> list, E key)
<E> int binarySearch(List<? extends E> list, E key,
                     Comparator<? super E> cmp))
```

```
Collections.sort(strList);
// Sorted in natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   // Successful:    2
out.println(Collections.binarySearch(strList, "bigfeet"));  // Unsuccessful: -3
out.println(Collections.binarySearch(strList, "bigmouth")); // Unsuccessful: -5
```

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   //  1
out.println(Collections.binarySearch(strList, "bigfeet"));  // -1 (INCORRECT)
out.println(Collections.binarySearch(strList, "bigmouth")); // -5 (INCORRECT)
```

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in reverse natural order:
out.println(Collections.binarySearch(strList, "bigger",
                                     Collections.reverseOrder())); //  1
```

```
out.println(Collections.binarySearch(strList, "bigfeet",
                                     Collections.reverseOrder())); // -3
out.println(Collections.binarySearch(strList, "bigmouth",
                                     Collections.reverseOrder())); // -1
```

```
int indexOfSubList(List<?> source, List<?> target)
int lastIndexOfSubList(List<?> source, List<?> target)
```

```
<E extends Object & Comparable<? super E>>
    E max(Collection<? extends E> c)
<E> E max(Collection<? extends E> c, Comparator<? super E> cmp)
<E extends Object & Comparable<? super E>>
    E min(Collection<? extends E> c)
<E> E min(Collection<? extends E> cl, Comparator<? super E> cmp)
```

```
// Defined in the List<E> interface.
void replaceAll(UnaryOperator<E> operator)
```

```
// Defined in the Collections class.
<E> boolean replaceAll(List<E> list, E oldVal, E newVal) // Collections
```

```
// Before: [biggest, big, bigger, Bigfoot]
strList.replaceAll(str -> str.toUpperCase());      // (1a)
strList.replaceAll(String::toUpperCase);           // (1b)
// After: [BIGGEST, BIG, BIGGER, BIGFOOT]
// Before: [eye, level, radar, road]
Collections.replaceAll(palindromes, "road", "anna");
// After: [eye, level, radar, anna]
```

```
<E> boolean addAll(Collection<? super E> collection, E... elements)
```

```
<E> void copy(List<? super E> destination, List<? extends E> source)
```

```
<E> void fill(List<? super E> list, E element)
```

```
<E> List<E> nCopies(int n, E element)
```

```
List<Integer> intList = new ArrayList<>();            // []
Collections.addAll(intList, 9, 1, 1);                 // (1) Varargs
// After: [9, 1, 1]
Collections.addAll(intList, new Integer[] {1, 1, 9}); // (2) An array of Integers
```

```
// After: [9, 1, 1, 1, 1, 9]
Collections.addAll(intList, new int[] {1, 9, 1});     // (3) Compile-time error!
```

```
List<String> dest = new ArrayList<>();
Collections.addAll(dest, "one", "two", "three", "four");// [one, two, three, four]
List<String> src = new ArrayList<>();
Collections.addAll(src, "I", "II", "III");               // [I, II, III]
Collections.copy(dest, src);                             // [I, II, III, four]
```

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "liberty", "equality", "fraternity");
// Before: [liberty, equality, fraternity]
Collections.fill(strList, "CENSORED");                  // (1)
// After: [CENSORED, CENSORED, CENSORED]
```

```
for (int i = 0; i < 5; ++i)
  System.out.printf("%d %s%n", i, Collections.nCopies(i, "*"));
```

```
void sort(type[] array)
void sort(type[] array, int fromIndex, int toIndex)
```

```
<E> void sort(E[] array, Comparator<? super E> cmp)
<E> void sort(E[] array, int fromIndex, int toIndex,
                  Comparator<? super E> cmp)
```

```
String[] strArray =  {"biggest", "big", "bigger", "Bigfoot"};
Arrays.sort(strArray);                                         // Natural order
Arrays.sort(strArray, Comparator.comparing(String::length));   // Length order
Arrays.sort(strArray, Collections.reverseOrder());   // Reverse natural order
Arrays.sort(strArray, String.CASE_INSENSITIVE_ORDER);// Case insensitive order
Arrays.sort(strArray,                        // Reverse case insensitive order
            Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

```
Before sorting:                                [biggest, big, bigger, Bigfoot]
After sorting in natural order:                [Bigfoot, big, bigger, biggest]
After sorting in length order:                 [big, bigger, Bigfoot, biggest]
After sorting in reverse natural order:        [biggest, bigger, big, Bigfoot]
After sorting in case insensitive order:       [big, Bigfoot, bigger, biggest]
After sorting in reverse case insensitive order: [biggest, bigger, Bigfoot, big]
```

```
int[] intArray = {5, 3, 7, 1};                  // int
Arrays.sort(intArray);                          // (1) Natural order: [1, 3, 5, 7]

Object[] objArray1 = {"I", "am", "OK"};         // String
Arrays.sort(objArray1);                         // (2) Natural order: [I, OK, am]

Comparable<Integer>[] comps = new Integer[] {5, 3, 7, 1}; // Integer
Arrays.sort(comps, 1, 4, Collections.reverseOrder());// (3) Reverse natural order:
                                                      //     [5, 7, 3, 1]

Object[] objArray2 = {23, 3.14, "ten"};         // Not mutually comparable
//  Arrays.sort(objArray2);                     // (4) ClassCastException

Number[] numbers = {23, 3.14, 10L};             // Not mutually comparable
//  Arrays.sort(numbers);                       // (5) ClassCastException
```

```
int binarySearch(type[] array, type key)
int binarySearch(type[] array, int fromIndex, int toIndex, type key)
```

```
<E> int binarySearch(E[] array, E key, Comparator<? super E> c)
<E> int binarySearch(E[] array, int fromIndex, int toIndex, E key,
                          Comparator<? super E> c)
```

```
Arrays.sort(strArray);
// Sorted according to natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Arrays.binarySearch(strArray, "bigger"));   // Successful:    2
out.println(Arrays.binarySearch(strArray, "bigfeet"));  // Unsuccessful: -3
out.println(Arrays.binarySearch(strArray, "bigmouth")); // Unsuccessful: -5
out.println(Arrays.binarySearch(strArray, "bigger",
                                Collections.reverseOrder()));  //  -1 (INCORRECT)
```

```
out.println(Arrays.binarySearch(strArray, 4)); // Key: 4 => ClassCastException
```

```
// Sorted int array (natural order): [1, 3, 5, 7]
out.println(Arrays.binarySearch(intArray, 4.5));// Key: 4.5 => Compile-time error!
```

`boolean equals(`*type*`[] a, `*type*`[] b)`

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Equals: true
```

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Equals: false
```

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Equals: false
```

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
...
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketB)); // true
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketC)); // false
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketE)); // false
out.println(Arrays.equals(null, null));             // Ambiguous method call.
                                                    //   Compile-time error!
out.println(Arrays.equals((String[]) null, null));  // true
```

```
int compare(type[] a, type[] b)
<T extends Comparable<? super T>> int compare(T[] a, T[] b)
```

```
int[] diceA = { 5, 2, 6, 3 };
int[] diceB = { 5, 2, 6, 3 };
int[] diceC = { 5, 2, 3, 6 };
int[] diceD = { 5, 2, 3 };
int[] diceE = { 5, 6 };
...
out.println("Compare value: " + Arrays.compare(diceA, diceD)); // 1
out.println("Compare value: " + Arrays.compare(diceA, diceE)); // -1
out.println("Compare value: " + Arrays.compare(diceA, diceB)); // 0
out.println("Compare value: " + Arrays.compare(diceC, diceD)); // 1
```

```
int mismatch(type[] a, type[] b)
```

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Common prefix: [oranges, apples]
Mismatch index: 2
```

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketG ==> [apples]
Common prefix: []
Mismatch index: 0
```

```
Index                 0        1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Prefix: [oranges, apples, plums, kiwi]
Mismatch value: -1
```

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Proper prefix: [oranges, apples]
Mismatch index: 2
```

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
String[] fruitBasketG = { "apples" };
...
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketC)); //2
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketG)); //0
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketB)); //-1
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketE)); //2
```

```
String toString(type[] array)
String deepToString(Object[] array)
```

```
void fill(type[] array, type value)
void fill(type[] array, int fromIndex, int toIndex, type value)
```

```
for (int i = 0; i < 5; ++i) {
  char[] bar = new char[i];
  Arrays.fill(bar, '*');
  System.out.printf("%d %s%n", i, Arrays.toString(bar));
}
```

```
char[][] twoDimArr = new char[2][2];
Arrays.fill(twoDimArr[0], '*');
Arrays.fill(twoDimArr[1], '*');
System.out.println(Arrays.deepToString(twoDimArr));    // [[*, *], [*, *]]
```

```
<T> void setAll(T[] array, IntFunction<? extends T> generator)
void setAll(int[] array, IntUnaryOperator generator)
void setAll(long[] array, IntToLongFunction generator)
void setAll(double[] array, IntToDoubleFunction generator)
```

```
String[] strArr = new String[5];
Arrays.setAll(strArr, i -> "Str@" + i); // [Str@0, Str@1, Str@2, Str@3, Str@4]

int[] intArr = new int[4];
Arrays.setAll(intArr, i -> i * i);      // [0, 1, 4, 9]
```

```
import java.util.*;
public class WhatIsThis {
  public static void main(String[] args) {
    List<StringBuilder> list = new ArrayList<>();
    list.add(new StringBuilder("B"));
    list.add(new StringBuilder("A"));
    list.add(new StringBuilder("C"));
    list.sort(Collections.reverseOrder());
    System.out.println(list.subList(1,2));
  }
}
```

```
import java.util.*;
public class InitOnly {
  public static void main(String[] args) {
    List<String> list1 = new ArrayList<>();
    Collections.addAll(list1, "Hi", "Hello");
    Collections.addAll(list1, "Howdy");
    System.out.println(list1);                       // (1)

    List<String> list2 = new ArrayList<>(list1);
    list2 = Arrays.asList("Hi", "Hello");
    list2 = Arrays.asList("Howdy");
    System.out.println(list2);                       // (2)
    List<String> list3 = new ArrayList<>();
    list3.addAll(list1);
    System.out.println(list3);                              // (3)

    List<String> list4 = new ArrayList<>(list2);
    System.out.println(list4);                              // (4)
  }
}
```

```
import java.util.Arrays;
public class GetThatIndex {
  public static void main(String[] args) {
    if (args.length != 1) return;
    printIndex(args[0]);
  }

  public static void printIndex(String key) {
    String[] strings = {"small", "smaller", "smallest", "tiny"};
    System.out.println(Arrays.binarySearch(strings, key));
  }
}
```

```
import java.util.*;
public class Q36 {
  public static void main(String[] args) {
    String s1 = "London";
    String s2 = "Bergen";
    String s3 = "Saratov";
    Deque<String> trip = new ArrayDeque<>();
    String r1 = trip.pollFirst();
    trip.offerFirst(s1);
    trip.offerFirst(s2);
    String r2 = trip.pollFirst();
    String r3 = trip.peekFirst();
    trip.offerLast(s3);
    trip.offerLast(s1);
    String r4 = trip.pollLast();
    String r5 = trip.peekLast();
    System.out.println(r1 + " " + r2 + " " + r3 + " " + r4 + " " + r5);
  }
}
```

```
import java.util.*;
public class Q37 {
  public static void main(String[] args) {
    Integer x = 1, y = 2, z = 3;
    Set<Integer> coordinates = new TreeSet<>();
    coordinates.add(x);
    coordinates.add(y);
    coordinates.add(y);
    coordinates.add(z);
    coordinates.remove(x);
    System.out.print(coordinates);
  }
}
```

```
import java.util.*;
public class Q38 {
  public static void main(String[] args) {
    List<Integer> prices = new ArrayList<>();
    prices.add(1);
    prices.add(2);
    prices.add(2, null);
    prices.add(3, 3);
    prices.add(2, 4);
    prices.set(2, 3);
    prices.remove(2);
    prices.add(2, 2);
    System.out.print(prices);
  }
}
```

```
// Loop-based solution:
List<String> values = List.of("2001", "1999", "2021");
for (String s : values) {
  Year y = Year.parse(s);
  if (y.isAfter(Year.of(2000))) {
    System.out.print(s + " ");                          // 2001 2021
  }
}

// Stream-based solution:
List<String> values2 = List.of("2001", "1999", "2021");
values2.stream()                                          // (1)
       .map(s -> Year.parse(s))                           // (2)
       .filter(y -> y.isAfter(Year.of(2000)))             // (3)
       .forEach(y -> System.out.print(y + " "));    // (4) 2001 2021
```

```
// The different genres in music.
public enum Genre {POP, JAZZ, OTHER}
```

```
import java.time.Year;
import java.util.Comparator;
import java.util.List;

/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Comparable<CD> {

  // Additional get methods:
  public boolean isPop()   { return this.genre == Genre.POP; }
  public boolean isJazz()  { return this.genre == Genre.JAZZ; }
  public boolean isOther() { return this.genre == Genre.OTHER; }

  // Provide own implementation of the toString() method.
  @Override public String toString() {
    return String.format("<%s, \"%s\", %d, %s, %s>",
        this.artist, this.title, this.noOfTracks, this.year, this.genre);
  }

  /** Compare by artist, by title, by number of tracks, by year, and by genre. */
  @Override public int compareTo(CD other) {
    return Comparator.comparing(CD::artist)
                     .thenComparing(CD::title)
                     .thenComparing(CD::noOfTracks)
                     .thenComparing(CD::year)
                     .thenComparing(CD::genre)
                     .compare(this, other);
  }

  // Some ready-made CDs.
  public static final CD cd0
      = new CD("Jaav",      "Java Jive",       8,  Year.of(2017), Genre.POP);
  public static final CD cd1
      = new CD("Jaav",      "Java Jam",        6,  Year.of(2017), Genre.JAZZ);
  public static final CD cd2
      = new CD("Funkies",   "Lambda Dancing",  10, Year.of(2018), Genre.POP);
  public static final CD cd3
      = new CD("Genericos", "Keep on Erasing", 8,  Year.of(2018), Genre.JAZZ);
  public static final CD cd4
      = new CD("Genericos", "Hot Generics",    10, Year.of(2018), Genre.JAZZ);
  // A fixed-size list of CDs.
  public static final List<CD> cdList = List.of(cd0, cd1, cd2, cd3, cd4);

  // An array of CDs.
  public static final CD[] cdArray = {cd0, cd1, cd2, cd3, cd4};
}
```

```java
import java.util.List;

public final class CDAdmin {
  public static void main(String[] args) {
    List<CD> cdList = CD.cdList;
    System.out.println("     Artist    Title           No. Year Genre");
    for(int i = 0; i < cdList.size(); ++i) {
      CD cd = cdList.get(i);
      String cdToString = String.format("%-10s%-16s%-4d%-5s%-5s",
          cd.artist(), cd.title(), cd.noOfTracks(),
          cd.year(), cd.genre());
      System.out.printf("cd%d: %s%n", i, cdToString);
    }
  }
}
```

```
     Artist    Title           No. Year Genre
     Artist    Title           No. Year Genre
cd0: Jaav      Java Jive       8   2017 POP
cd1: Jaav      Java Jam        6   2017 JAZZ
cd2: Funkies   Lambda Dancing  10  2018 POP
cd3: Genericos Keep on Erasing 8   2018 JAZZ
cd4: Genericos Hot Generics    10  2018 JAZZ
```

```
Stream<CD> stream1 = cdList.stream();            // (1a) Stream creation.
Stream<CD> stream2 = stream1.filter(CD::isPop);  // (2a) Intermediate operation.
List<CD> popCDs2 = stream2.toList();             // (3a) Terminal operation.
```

```
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]
```

```
import java.util.List;
import java.util.stream.Stream;

public class StreamPipeLine {
  public static void main(String[] args) {

    List<CD> cdList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd3, CD.cd4);

    // (A) Query to create a list of all CDs with pop music.
    List<CD> popCDs = cdList.stream()              // (1) Stream creation.
        .filter(CD::isPop)                         // (2) Intermediate operation.
        .toList();                                 // (3) Terminal operation.
    System.out.println(popCDs);

    // (B) Equivalent to (A).
    Stream<CD> stream1 = cdList.stream();          // (1a) Stream creation.
    Stream<CD> stream2 = stream1.filter(CD::isPop);// (2a) Intermediate operation.
    List<CD> popCDs2 = stream2.toList();           // (3a) Terminal operation.
    System.out.println(popCDs2);
  }
}
```

```
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]
[<Jaav, "Java Jive", 8, 2017, POP>, <Funkies, "Lambda Dancing", 10, 2018, POP>]
```

```
static <T> Stream<T> empty()
static <T> Stream<T> of(T t)
static <T> Stream<T> of(T... varargs)
```

`static <T> Stream<T> ofNullable(T t)`	Only in the `Stream<T>` interface.

```
static <T> Stream<T> generate(Supplier<T> supplier)
static <T> Stream<T> iterate(T s, UnaryOperator<T> uop)
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

```
Stream<CD> cdStream = Stream.empty();                    // Empty stream of CD.
System.out.println("Count: " + cdStream.count());        // Count: 0
IntStream iStream = IntStream.empty();                   // Empty stream of int.
System.out.println("Count: " + iStream.count());         // Count: 0
```

```
// From specified objects.
Stream<CD> cdStream1 = Stream.of(CD.cd0);                 // (1) Single-arg call.
Stream<CD> cdStream2 = Stream.of(CD.cd0, CD.cd1, CD.cd2);  // (2) Varargs call.
cdStream2.filter(CD::isPop)                               // (3)
         .forEach(cd -> System.out.println(cd.title()));   // (4)
```

```
// From specified numeric values.
Stream<Integer> integerStream1 = Stream.of(2017, 2018, 2019);         // (1)
Stream<? extends Number> numStream = Stream.of(100, 3.14D, 5050L);   // (2)
IntStream intStream1 = IntStream.of(2017, 2018, 2019);               // (3)
DoubleStream doubleStream = DoubleStream.of(100, 3.14D, 5050L);      // (4)
```

```
// From an array of CDs.
Stream<CD> cdStream3 = Stream.of(CD.cdArray);                // (1)
Stream<CD> cdStream4 = Arrays.stream(CD.cdArray);            // (2)

// From an array of Integer.
Integer[] integerArray = {2017, 2018, 2019};                 // (3)
Stream<Integer> integerStream2 = Stream.of(integerArray);    // (4)
Stream<Integer> integerStream3 = Arrays.stream(integerArray); // (5)

// From an array of int.
int[] intArray = {2017, 2018, 2019};                         // (6)
Stream<int[]> intArrayStream = Stream.of(intArray);          // (7) Size is 1.
IntStream intStream2 = IntStream.of(intArray);               // (8) Size is 3.
IntStream intStream3 = Arrays.stream(intArray);              // (9) Size is 3.
```

```
IntSupplier supplier = () -> (int) (6.0 * Math.random()) + 1;  // (1)
IntStream diceStream = IntStream.generate(supplier);           // (2)
diceStream.limit(5)                                            // (3)
          .forEach(i -> System.out.print(i + " "));            // (4) 2 4 5 2 6
```

```
IntUnaryOperator uop = n -> n + 2;                // (1)
IntStream oddNums = IntStream.iterate(1, uop);    // (2)
oddNums.limit(5)
       .forEach(i -> System.out.print(i + " "));  // 1 3 5 7 9
```

```
Stream.iterate("ba", b -> b + "na")
       .limit(5)
       .forEach(System.out::println);
```

```
Set<String> strSet1                                                          // (1)
    = Set.of("All", " objects", " are", " equal");
Set<String> strSet2                                                          // (2)
    = Set.of(" but", " some", " are", " more", " equal", " than", " others.");
Stream<String> unorderedStream1 = strSet1.stream();                          // (3)
Stream<String> unorderedStream2 = strSet2.stream();                          // (4)
Stream.concat(unorderedStream1, unorderedStream2)                            // (5)
      .forEachOrdered(System.out::print);                                    // (6)
// objectsAll equal are some are others. than equal more but
```

```
Stream<String> orderedStream1 = Stream.of("All", " objects",                // (1)
                                          " are", " equal");
Stream<String> orderedStream2 = Stream.of(" but", " some", " are", " more", // (2)
                                          " equal", " than", " others.");
Stream.concat(orderedStream1, orderedStream2)                               // (3)
      .forEachOrdered(System.out::print);                                   // (4)
// All objects are equal but some are more equal than others.
```

```
Stream<String> pStream1 = strSet1.stream().parallel();                // (1)
System.out.println("pStream1 is parallel: " + pStream1.isParallel()); // (2) true
Stream<String> rStream = Stream.concat(pStream1, strSet2.stream());   // (3)
System.out.println("rStream is parallel: " + pStream1.isParallel());  // (4) true
rStream.forEachOrdered(System.out::print);                            // (5)
// objectsAll equal are some are others. than equal more but
```

```
default Stream<E> stream()
default Stream<E> parallelStream()
```

```
List<CD> listOfCDS = new ArrayList<>(List.of(CD.cd0, CD.cd1));       // (1)
Stream<CD> cdStream = listOfCDS.stream();                             // (2)
listOfCDS.add(CD.cd2);                                                // (3)
System.out.println(cdStream.count());                                 // (4) 3
// System.out.println(cdStream.count());             // (5) IllegalStateException
```

```
Map<Integer, String> dataMap = new HashMap<>();                         // (1)
dataMap.put(1, "en"); dataMap.put(2, "to");
dataMap.put(3, "tre"); dataMap.put(4, "fire");
long numOfEntries = dataMap
    .entrySet()                                                         // (2)
```

```
        .stream()                                                           // (3)
        .count();                                                           // (4) 4
```

```
Stream<CD> cdStream = Arrays.stream(cdArray, 1, 4);            // (1)
long noOfElements = cdStream.count();                          // (2) 3
```

```
static <T> Stream<T> stream(T[] array)
static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)
static NumTypeStream stream(numtype[] array)
static NumTypeStream stream(numtype[] array,
                                   int startInclusive, int endExclusive)
```

```
static NumTypeStream range(numtype startInclusive, numtype endExclusive)
static NumTypeStream rangeClosed(numtype startInclusive,
                                 numtype endInclusive)
```

```
for (int i = startInclusive; i < endExclusive; i++) {
  // Loop body.
}
```

```
IntStream.range(0, CD.cdArray.length)                               // (1)
         .forEach(i -> System.out.println(cdArray[CD.cdArray.length - 1 - i]));
```

```
int divisor = 5;
int start = 2000, end = 3000;
long divisibles = IntStream
    .rangeClosed(start, end)                                           // (1)
    .filter(number -> number % divisor == 0)                           // (2)
    .count();                                                          // (3)
System.out.println(divisibles);                                        // 201
```

```
int first = 10, len = 8;
int[] intArray = IntStream.range(first, first + len).toArray();          // (1)
System.out.println(intArray.length + ": " + Arrays.toString(intArray));
//8: [10, 11, 12, 13, 14, 15, 16, 17]
```

```
IntStream.rangeClosed(1, 10)                                 // Outer range.
         .forEach(i -> IntStream.rangeClosed(1, 10)          // Inner range.
                           .forEach(j -> System.out.printf("%2d * %2d = %2d%n",
                                                           i, j, i * j)));
}
```

```
NumTypeStream numtypes()
NumTypeStream numtypes(numtype randomNumberOrigin,
                       numtype randomNumberBound)
NumTypeStream numtypes(long streamSize)
NumTypeStream numtypes(long streamSize, numtype randomNumberOrigin,
                       numtype randomNumberBound)
```

```
Random rng = new Random();                                    // (1)
```

```
IntStream iStream = rng.ints();                 // (2) Unlimited, any int value
int[] intArray = iStream.limit(3).toArray();    // Limits size to 3
//[-1170441471, 1070948914, 264046613]
```

```
intArray = rng.ints(3).toArray();                // (3) Size 3, any int value
//[1011448344, -974832344, 816809715]
```

```
intArray = rng.ints(1, 7)                           // (4) Unlimited, [1, 6]
              .limit(3)                             // Limits size to 3
              .toArray();                           // [5, 2, 4]

intArray = rng.ints(3, 1, 7)                        // (5) Size 3, [1, 6]
              .toArray();                           // [1, 4, 6]
```

```
DoubleStream dStream = rng.doubles(3);            // (6) Size 3, [0.0, 1.0)
double[] dArray = dStream.toArray();
//[0.9333794789872794, 0.7037326827186609, 0.2839257522887708]
```

```
String strSource = "banananana";
IntStream iStream = strSource.chars();                   // (1)
iStream.forEach(i -> System.out.print(i + " "));         // Prints ints.
// 98 97 110 97 110 97 110 97 110 97

strSource.chars()
         .mapToObj(i -> (char)i)                         // (2) Stream<Character>
         .forEach(System.out::print);                    // Prints chars.
// banananana
```

```
String inputLines = "Wise men learn from their mistakes.\n"               // (1)
                  + "But wiser men learn from the mistakes of others.\n"
                  + "And fools just carry on.";
Stream<String> lStream = inputLines.lines();                             // (2)
lStream.filter(l -> l.contains("mistakes")).forEach(System.out::println);   // (3)
```

Wise men learn from their mistakes.
But wiser men learn from the mistakes of others.

```
try ( FileReader fReader = new FileReader("CD_Data.txt");          // (1)
      BufferedReader bReader = new BufferedReader(fReader);        // (2)
      Stream<String> lStream = bReader.lines() ) {                 // (3)
  System.out.println("Number of lines: " + lStream.count());       // (4) 13
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

```
Path path = Paths.get("CD_Data.txt");                              // (1)
try (Stream<String> lStream = Files.lines(path)) {                 // (2)
  System.out.println("Number of lines: " + lStream.count());      // (3) 13
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

```
static Stream<String> lines(Path path)
static Stream<String> lines(Path path, Charset cs)
```

```
 import java.util.List;

public class StreamOps {
  public static void main(String[] args) {

    // Query: Create a list with titles of pop music CDs.

    // (1) Stream Mapping:
    List<CD> cdList1 = CD.cdList;
    List<String> popCDs1 = cdList1
        .stream()                      // Initial stream:         Stream<CD>
        .filter(CD::isPop)             // Intermediate operation: Stream<CD>
        .map(CD::title)                // Intermediate operation: Stream<String>
        .toList();                     // Terminal operation: List<String>
    System.out.println("Pop music CDs: " + popCDs1);
    System.out.println();

    // (2) Lazy Evaluation:
    List<CD> cdList2 = CD.cdList;
    List<String> popCDs2 = cdList2
        .stream()                      // Initial stream:          Stream<CD>
        .filter(cd -> {                 // Intermediate operation:  Stream<CD>
           System.out.println("Filtering: " + cd                      // (3)
                            + (cd.isPop() ? " is pop CD." : " is not pop CD."));
           return cd.isPop();
         })
        .map(cd -> {                    // Intermediate operation: Stream<String>
           System.out.println("Mapping: " + cd.title());          // (4)
           return cd.title();
         })
        .toList();                      // Terminal operation: List<String>
    System.out.println("Pop music CDs: " + popCDs2);
  }
}
```

```
Pop music CDs: [Java Jive, Lambda Dancing]

Filtering: <Jaav, "Java Jive", 8, 2017, POP> is pop CD.
Mapping: Java Jive
Filtering: <Jaav, "Java Jam", 6, 2017, JAZZ> is not pop CD.
Filtering: <Funkies, "Lambda Dancing", 10, 2018, POP> is pop CD.
Mapping: Lambda Dancing
Filtering: <Genericos, "Keep on Erasing", 8, 2018, JAZZ> is not pop CD.
Filtering: <Genericos, "Hot Generics", 10, 2018, JAZZ> is not pop CD.
Pop music CDs: [Java Jive, Lambda Dancing]
```

```
import java.util.List;

public final class OrderOfOperations {
  public static void main(String[] args) {

    List<CD> cdList = CD.cdList;

    // Map before skip.
    List<String> cdTitles1 = cdList
        .stream()                    // (1)
        .map(cd -> {                 // Map applied to all elements.
           System.out.println("Mapping: " + cd.title());
           return cd.title();
         })
        .skip(3)                     // Skip afterwards.
        .toList();
    System.out.println(cdTitles1);
    System.out.println();

    // Skip before map preferable.
    List<String> cdTitles2 = cdList
        .stream()                    // (2)
        .skip(3)                     // Skip first.
        .map(cd -> {                 // Map not applied to the first 3 elements.
           System.out.println("Mapping: " + cd.title());
           return cd.title();
         })
        .toList();
    System.out.println(cdTitles2);
  }
}
```

```
Mapping: Java Jive
Mapping: Java Jam
Mapping: Lambda Dancing
Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]

Mapping: Keep on Erasing
Mapping: Hot Generics
[Keep on Erasing, Hot Generics]
```

```
// Filtering using a predicate.
Stream<T> filter(Predicate<? super T> predicate)
```

```
// Taking and dropping elements using a predicate.
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

```
// Selecting distinct elements.
Stream<T> distinct()
```

```
List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
System.out.println("Query 1b: " + popularCDs2);
//Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
```

```
// Ordered stream:
Stream.of(1, 3, 5, 7, 8, 9, 11)                        // (1)
```

```
      .takeWhile(n -> n % 2 != 0)                 // Takes longest prefix: 1 3 5 7
      .forEach(n -> System.out.print(n + " ")); // 1 3 5 7

Stream.of(1, 3, 5, 7, 8, 9, 11)                 // (2)
      .dropWhile(n -> n % 2 != 0)                 // Drops longest prefix:  1 3 5 7
      .forEach(n -> System.out.print(n + " ")); // 8 9 11
```

```
// Unordered stream:
Set<Integer> iSeq = Set.of(1, 9, 4, 3, 7);       // (3)
iSeq.stream()
    .takeWhile(n -> n % 2 != 0)                  // Takes any subset of elements.
    .forEach(n -> System.out.print(n + " "));   // Nondeterministic: 1 9 7

iSeq.stream()
    .dropWhile(n -> n % 2 != 0)                  // Drops any subset of elements.
    .forEach(n -> System.out.print(n + " "));   // Nondeterministic: 4 3
```

```
// All match in ordered stream:                    (4)
Stream.of(1, 3, 5, 7, 9, 11)
      .takeWhile(n -> n % 2 != 0)               // Takes all elements.
      .forEach(n -> System.out.print(n + " ")); // Ordered: 1 3 5 7 9 11

Stream.of(1, 3, 5, 7, 9, 11)
      .dropWhile(n -> n % 2 != 0)               // Drops all elements.
      .forEach(n -> System.out.print(n + " ")); // Empty stream

// All match in unordered stream:                  (5)
Set<Integer> iSeq2 = Set.of(1, 9, 3, 7, 11, 5);
iSeq2.stream()
     .takeWhile(n -> n % 2 != 0)                // Takes all elements.
     .forEach(n -> System.out.print(n + " "));  // Unordered: 9 11 1 3 5 7

iSeq2.stream()
     .dropWhile(n -> n % 2 != 0)                // Drops all elements.
     .forEach(n -> System.out.print(n + " "));  // Empty stream
```

```
// No match in ordered stream:                        (6)
Stream.of(2, 4, 6, 8, 10, 12)
      .takeWhile(n -> n % 2 != 0)                  // Takes no elements.
      .forEach(n -> System.out.print(n + " ")); // Empty stream

Stream.of(2, 4, 6, 8, 10, 12)
      .dropWhile(n -> n % 2 != 0)                  // Drops no elements.
      .forEach(n -> System.out.print(n + " ")); // Ordered: 2 4 6 8 10 12
```

```
// No match in unordered stream:                    (7)
Set<Integer> iSeq3 = Set.of(2, 10, 8, 12, 4, 6);
iSeq3.stream()
     .takeWhile(n -> n % 2 != 0)                 // Takes no elements.
     .forEach(n -> System.out.print(n + " "));  // Empty stream

iSeq3.stream()
     .dropWhile(n -> n % 2 != 0)                 // Drops no elements.
     .forEach(n -> System.out.print(n + " "));  // Unordered: 8 10 12 2 4 6
```

```
List<CD> resultList = Stream
    .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
    .toList();
System.out.println(CD.cdList.equals(resultList));            // true
```

```
List<CD> substream = CD.cdList
    .stream()
    .skip(1)
    .limit(3)
    .toList();
System.out.println("Query 5: " + substream);
// Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
//           <Funkies, "Lambda Dancing", 10, 2018, POP>,
//           <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

```
import java.time.Year;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Stream;

public final class Filtering {
  public static void main(String[] args) {

    // Query 1: Find CDs whose titles are in the set of popular CD titles.
    Set<String> popularTitles = Set.of("Java Jive", "Java Jazz", "Java Jam");

    // Using Stream.filter().
    List<CD> popularCDs1 = CD.cdList
        .stream()
        .filter(cd -> popularTitles.contains(cd.title()))
        .toList();
    System.out.println("Query 1a: " + popularCDs1);
    // Using Collection.removeIf().
    List<CD> popularCDs2 = new ArrayList<>(CD.cdList);
    popularCDs2.removeIf(cd -> !(popularTitles.contains(cd.title())));
    System.out.println("Query 1b: " + popularCDs2);

    // Query 2: Create a list of unique CDs with pop music.
    List<CD> miscCDList = List.of(CD.cd0, CD.cd0, CD.cd1, CD.cd0);
    List<CD> uniquePopCDs1 = miscCDList
        .stream()
        .filter(CD::isPop)
        .distinct()                                // distinct() after filter()
        .toList();
    System.out.println("Query 2: " + uniquePopCDs1);

    // Query 3a: Create a list of jazz CDs, after skipping the first two CDs.
    List<CD> jazzCDs1 = CD.cdList
        .stream()
        .skip(2)                                   // skip() before filter().
        .filter(CD::isJazz)
        .toList();
    System.out.println("Query 3a: " + jazzCDs1);

    // Query 3b: Create a list of jazz CDs, but skip the first two jazz CDs.
    List<CD> jazzCDs2 = CD.cdList                  // Not equivalent to Query 3
        .stream()
        .filter(CD::isJazz)
        .skip(2)                                   // skip() after filter().
        .toList();
    System.out.println("Query 3b: " + jazzCDs2);

    // Query 4: Create a list with the first 2 CDs that were released in 2018.
    List<CD> twoFirstCDs2018 = CD.cdList
        .stream()
        .filter(cd -> cd.year().equals(Year.of(2018)))
        .limit(2)
        .toList();
    System.out.println("Query 4: " + twoFirstCDs2018);

    // limit(n) and skip(n) are complementary.
    List<CD> resultList = Stream
        .concat(CD.cdList.stream().limit(2), CD.cdList.stream().skip(2))
        .toList();
    System.out.println(CD.cdList.equals(resultList));

    // Query 5: Process a substream by skipping 1 and limiting the size to 3.
    List<CD> substream = CD.cdList
        .stream()
        .skip(1)
        .limit(3)
        .toList();
    System.out.println("Query 5: " + substream);
  }
}
```

```
Query 1a: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
Query 1b: [<Jaav, "Java Jive", 8, 2017, POP>, <Jaav, "Java Jam", 6, 2017, JAZZ>]
Query 2: [<Jaav, "Java Jive", 8, 2017, POP>]
Query 3a: [<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
           <Genericos, "Hot Generics", 10, 2018, JAZZ>]
Query 3b: [<Genericos, "Hot Generics", 10, 2018, JAZZ>]
Query 4: [<Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
true
Query 5: [<Jaav, "Java Jam", 6, 2017, JAZZ>,
          <Funkies, "Lambda Dancing", 10, 2018, POP>,
          <Genericos, "Keep on Erasing", 8, 2018, JAZZ>]
```

```
Stream<T> peek(Consumer<? super T> action)
```

```
import java.util.List;

public final class OrderOfOperationsWithPeek {
  public static void main(String[] args) {
    System.out.println("map() before skip():");
    List<String> cdTitles1 = CD.cdList
        .stream()
        .map(CD::title)
        .peek(t -> System.out.println("After map: " + t))
        .skip(3)
        .peek(t -> System.out.println("After skip: " + t))
        .toList();
    System.out.println(cdTitles1);
    System.out.println();

    System.out.println("skip() before map():");            // Preferable.
    List<String> cdTitles2 = CD.cdList
        .stream()
        .skip(3)
        .peek(cd -> System.out.println("After skip: " + cd))
        .map(CD::title)
        .peek(t -> System.out.println("After map: " + t))
        .toList();
    System.out.println(cdTitles2);
  }
}
```

```
map() before skip():
After map: Java Jive
After map: Java Jam
After map: Lambda Dancing
After map: Keep on Erasing
After skip: Keep on Erasing
After map: Hot Generics
After skip: Hot Generics
[Keep on Erasing, Hot Generics]

skip() before map():
After skip: <Genericos, "Keep on Erasing", 8, 2018, JAZZ>
After map: Keep on Erasing
After skip: <Genericos, "Hot Generics", 10, 2018, JAZZ>
After map: Hot Generics
[Keep on Erasing, Hot Generics]
```

```
Stream<String> titles = CD.cdList
         .stream()                      // Input stream: Stream<CD>.
         .map(CD::title);               // Lambda expression: cd -> cd.title()
```

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

```
// Converting Stream<T> to a Numeric Stream
IntStream    mapToInt(ToIntFunction<? super T> mapper)
LongStream   mapToLong(ToLongFunction<? super T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

```
CD.cdList.stream()                                            // Stream<CD>
         .map(CD::year)                                       // Stream<Year>
         .distinct()                                          // Stream<Year>
         .mapToInt(Year::getValue)                            // IntStream
         .forEach(System.out::println);                       // 2017
                                                              // 2018
```

```
int from = 0, to = 5;
IntStream.range(from, to)                 // [0, 5)
         .map(i -> to + from - 1 - i)     // Reverse the stream values
         .forEach(System.out::print);     // 43210
```

```
long sixes = IntStream
   .generate(() -> (int) (6.0 * Math.random()) + 1) // [1, 6]
   .limit(2000)                                      // Number of throws.
   .map(i -> i == 6 ? 1 : 0)              // Dice value mapped to 1 or 0.
   .sum();
```

```
<R> Stream<R> flatMap(
                    Function<? super T,? extends Stream<? extends R>> mapper)
```

```
IntStream    flatMapToInt(Function<? super T,? extends IntStream> mapper)
LongStream   flatMapToLong(Function<? super T,? extends LongStream> mapper)
DoubleStream flatMapToDouble(
                 Function<? super T,? extends DoubleStream> mapper)
```

```
int[][] twoDimArray = { {2017, 2018}, {1948, 1949} };
int[] intArray = Arrays
    .stream(twoDimArray)                       // (1) Stream<int[]>
    .flatMapToInt(row -> Arrays.stream(row)) // (2) mapper: int[] -> IntStream,
                                         // flattens Stream<IntStream> to IntStream.
    .toArray();                                // [2017, 2018, 1948, 1949]
```

```
// One-to-one
BiConsumer<CD, Consumer<String>> bcA = (cd, consumer) -> {            // (1)
  if (cd.genre() == Genre.POP) {                                      // (2)
    consumer.accept(String.format("%-15s: %s", cd.title(),            // (3)
                                  "*".repeat(cd.noOfTracks())));
  }
};

CD.cdList.stream()                                                    // (4)
         .mapMulti(bcA)                                               // (5)
         .forEach(System.out::println);
```

```
// One-to-many
List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1);
List<CD> cdList2 = List.of(CD.cd0, CD.cd1);
BiConsumer<List<CD>, Consumer<String>> bcB = (lst, consumer) -> {       // (1)
  for (CD cd : lst) {
    consumer.accept(cd.title());
  }
};
List<String> listOfCDTitles = Stream.of(cdList1, cdList2) // (2) Stream<List<CD>>
    .mapMulti(bcB)                                        // (3)
    .distinct()
    .toList();
System.out.println(listOfCDTitles);                       // [Java Jive, Java Jam]
```

```
List<String> listOfCDTitles2 = Stream.of(cdList1,cdList2) // (1) Stream<List<CD>>
//  .mapMulti((lst, consumer) -> {                    // (2a) Compile-time error!
//  .<String>mapMulti((lst, consumer) -> {                       // (2b) OK.
    .mapMulti((List<CD> lst, Consumer<String> consumer) -> {   // (2c) OK.
      for (CD cd : lst) {
        consumer.accept(cd.title());
      }
    })
    .distinct()
    .toList();
System.out.println(listOfCDTitles2);                  // [Java Jive, Java Jam]
```

```
default <R> Stream<R> mapMulti(
                  BiConsumer<? super T,? super Consumer<R>> mapper)
```

```
default IntStream
       mapMultiToInt(BiConsumer<? super T,? super IntConsumer> mapper)
default LongStream
       mapMultiToLong(BiConsumer<? super T,? super LongConsumer> mapper)
default DoubleStream
       mapMultiToDouble(BiConsumer<? super T,? super DoubleConsumer> mapper)
```

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> cmp)
```

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Sorting {
  public static void main(String[] args) {

    System.out.println("(1) Positional order in the array:");
    CD[] cdArray = CD.cdArray;
    System.out.println(Arrays.toString(cdArray));              // (1)

    System.out.println("(2) Positional order in the stream:");
    List<CD> cdsByPositionalOrder =                            // (2)
      Arrays.stream(cdArray)
            .toList();
    System.out.println(cdsByPositionalOrder);

    System.out.println("(3) Natural order:");
    List<CD> cdsByNaturalOrder =                               // (3)
      Arrays.stream(cdArray)
            .sorted()
            .toList();
    System.out.println(cdsByNaturalOrder);

    System.out.println("(4) Reversed natural order:");
    List<CD> cdsByRNO =                                        // (4)
      Arrays.stream(cdArray)
//          .sorted((c1, c2) -> -c1.compareTo(c2))
            .sorted(Comparator.reverseOrder())
            .toList();
    System.out.println(cdsByRNO);

    System.out.println("(5) Only Jazz CDs, ordered by title:");
    List<String> jazzCDsByTitle =                              // (5)
      Arrays.stream(cdArray)
            .filter(CD::isJazz)
//          .sorted((c1, c2) -> c1.title().compareTo(c2.title()))
            .sorted(Comparator.comparing(CD::title))
            .map(CD::title)
            .toList();
    System.out.println(jazzCDsByTitle);

    System.out.println("(6) No. of tracks >= 8, ordered by number of tracks:");
    List<CD> cds =                                             // (6)
      Arrays.stream(cdArray)
            .filter(d -> d.noOfTracks() >= 8)
//          .sorted((c1, c2) -> c1.noOfTracks() - c2.noOfTracks())
            .sorted(Comparator.comparing(CD::noOfTracks))
            .toList();
    System.out.println(cds);
  }
}
```

```
(1) Positional order in the array:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
(2) Positional order in the stream:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
(3) Natural order:
[<Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Jaav, "Java Jive", 8, 2017, POP>]
(4) Reversed natural order:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Jaav, "Java Jam", 6, 2017, JAZZ>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>]
(5) Only Jazz CDs, ordered by title:
[Hot Generics, Java Jam, Keep on Erasing]
(6) No. of tracks >= 8, ordered by number of tracks:
[<Jaav, "Java Jive", 8, 2017, POP>,
 <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
 <Funkies, "Lambda Dancing", 10, 2018, POP>,
 <Genericos, "Hot Generics", 10, 2018, JAZZ>]
```

```
//Query: Create a list with the first 2 Jazz CD titles.
List<String> first2JazzCDTitles = CD.cdList
    .stream()
    .unordered()                      // Don't care about ordering.
    .filter(CD::isJazz)
    .limit(2)
    .map(CD::title)
    .toList();                        // [Java Jam, Keep on Erasing]
```

```
Stream<CD> seqStream1
  = CD.cdList.stream().filter(CD::isPop);                         // Sequential
Stream<CD> seqStream2
  = CD.cdList.stream().sequential().filter(CD::isPop);            // Sequential
Stream<CD> seqStream3
  = CD.cdList.stream().parallel().filter(CD::isPop).sequential(); // Sequential
Stream<CD> paraStream1
  = CD.cdList.stream().parallel().filter(CD::isPop);              // Parallel
Stream<CD> paraStream2
  = CD.cdList.stream().filter(CD::isPop).parallel();              // Parallel
```

```
System.out.println(seqStream3.isParallel());                    // false
```

```
int totalNumOfTracks = CD.cdList
    .stream()                                               // (1) Stream<CD>
    .mapToInt(CD::noOfTracks)                               // (2) IntStream
    .sum();                                                 // 42
```

```
List<CD> cdList1 = List.of(CD.cd0, CD.cd1);
List<CD> cdList2 = List.of(CD.cd2, CD.cd3, CD.cd4);
int totalNumOfTracks =
    Stream.of(cdList1, cdList2)                      // Stream<List<CD>>
          .flatMapToInt(                             // (1)
              lst -> lst.stream()                    // Stream<CD>
                        .mapToInt(CD::noOfTracks))   // IntStream
                                                     // Stream<IntStream>,
                                                     //   flattened to IntStream.
          .sum();                                    // 42
```

```
IntStream.rangeClosed(1, 3)                          // (1) IntStream
         .mapToObj(n -> List.of(n, n*n))             // (2) Stream<List<Integer>>
         .forEach(p -> System.out.print(p + " "));   // [1, 1] [2, 4] [3, 9]
```

```
IntStream.rangeClosed(1, 3)                               // (3) IntStream
         .boxed()                                         // (4) Stream<Integer>
         .map(n -> List.of(n, n*n))                       // (5) Stream<List<Integer>>
         .forEach(p -> System.out.print(p + " "));   // [1, 1] [2, 4] [3, 9]
```

```
List<String> subListTitles = IntStream
    .rangeClosed(2, 3)                          // IntStream
    .mapToObj(i -> CD.cdList.get(i).title())  // Stream<String>
    .toList();                                  // [Lambda Dancing, Keep on Erasing]
```

```
IntStream.rangeClosed(1, 3)                          // IntStream
         .map(i -> i * i)                            // IntStream
         .forEach(n -> System.out.printf("%d ", n));  // 1 4 9
```

```
IntStream.rangeClosed(1, 3)                          // IntStream
         .flatMap(i -> IntStream.rangeClosed(1, 4))  // IntStream
         .forEach(n -> System.out.printf("%d ", n)); // 1 2 3 4 1 2 3 4 1 2 3 4
```

```
IntStream.rangeClosed(1, 3)                            // IntStream
         .mapToDouble(i -> Math.sqrt(i))               // DoubleStream
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

```
IntStream.rangeClosed(1, 3)                              // IntStream
         .asDoubleStream()                               // DoubleStream
         .map(d -> Math.sqrt(d))                         // DoubleStream
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

```
IntStream.rangeClosed(1, 3)                              // IntStream
         .boxed()                                        // Stream<Integer>
         .map(n -> Math.sqrt(n))                         // Stream<Double>
         .forEach(d -> System.out.printf("%.2f ", d));// 1.00 1.41 1.73
```

```
class Book {
  private Optional<String> optBlurb;

  public Optional<String> getOptBlurb() { return optBlurb; }

  //...
}
```

```
static <T> Optional<T> empty()
static <T> Optional<T> of(T nonNullValue)
static <T> Optional<T> ofNullable(T value)
```

```
Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
Optional<String> xblurb = Optional.of(null);   // NullPointerException
```

```
Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
Optional<String> noBlurb2 = Optional.ofNullable(null);   // Optional.empty()
Optional<String> noBlurb3 = Optional.empty();
```

```
// File: OptionalUsage.java
import java.util.Optional;

// A book can have an optional blurb.
class Book {
  private String bookName;
  private Optional<String> optBlurb;

  public String getBookName() { return bookName; }
  public Optional<String> getOptBlurb() { return optBlurb; }
  public Book(String bookName, Optional<String> optBlurb) {
    this.bookName = bookName;
    this.optBlurb = optBlurb;
  }
}

// A course can have an optional book.
class Course {
  private Optional<Book> optBook;
  public Optional<Book> getOptBook() { return optBook; }
  public Course(Optional<Book> optBook) { this.optBook = optBook; }
}

public class OptionalUsage {
  public static void main(String[] args) {

    // Creating an Optional:
    Optional<String> blurb0 = Optional.of("Java Programmers tell all!");
    //Optional<String> xblurb = Optional.of(null);   // NullPointerException
    Optional<String> blurb1 = Optional.ofNullable("Program like a Java Pro!");
    Optional<String> noBlurb2 = Optional.ofNullable(null);   // Optional.empty()
    Optional<String> noBlurb3 = Optional.empty();

    // Create some books:
    Book book0 = new Book("Embarrassing Exceptions", blurb0);
    Book book1 = new Book("Dancing Lambdas", noBlurb2);      // No blurb.

    // Querying an Optional:
    if (book0.getOptBlurb().isPresent()) {
      System.out.println(book0.getOptBlurb().get());//Java Programmers tell all!
    }

    book0.getOptBlurb()
         .ifPresent(System.out::println);           //Java Programmers tell all!

//  System.out.println(book1.getOptBlurb().get());  // NoSuchElementException

    String blurb = book0.getOptBlurb()
                        .orElse("No blurb");     // "Java Programmers tell all!"
    System.out.println(blurb);

    blurb = book1.getOptBlurb().orElse("No blurb");          // "No blurb"
    System.out.println(blurb);

    blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
    System.out.println(blurb);

    //blurb = book1.getOptBlurb()                            // RuntimeException
    //             .orElseThrow(() -> new RuntimeException("No blurb"));
  }
}
```

```
if (book0.getOptBlurb().isPresent()) {
  System.out.println(book0.getOptBlurb().get());    // Java Programmers tell all!
}
System.out.println(book1.getOptBlurb().get());      // NoSuchElementException
```

```
bookO.getOptBlurb().ifPresent(System.out::println); //Java Programmers tell all!
```

```
T get()
```

```
boolean isPresent()
void ifPresent(Consumer<? super T> consumer)
```

```
T orElse(T other)
T orElseGet(Supplier<? extends T> other)
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
                            throws X extends Throwable
```

```
String blurb = book0.getOptBlurb()
                    .orElse("No blurb");           // "Java Programmers tell all!"
```

```
blurb = book1.getOptBlurb().orElse("No blurb");    // "No blurb"
blurb = book1.getOptBlurb().orElseGet(() -> "No blurb"); // "No blurb"
blurb = book1.getOptBlurb()                         // RuntimeException
             .orElseThrow(() -> new RuntimeException("No blurb"));
```

```
static OptionalNumType empty()
static OptionalNumType of(numtype value)
```

```
boolean isPresent()
void    ifPresent(NumTypeConsumer consumer)
```

```
numtype orElse(numtype other)
numtype orElseGet(NumTypeSupplier other)
<X extends Throwable> numtype orElseThrow(Supplier<X> exceptionSupplier)
                                           throws X extends Throwable
```

```
// File: NumericOptionalUsage.java
import java.util.OptionalInt;

class Recipe {
  private String recipeName;
  private OptionalInt calories;    // Optional number of calories.

  public String getRecipeName() { return recipeName; }
  public OptionalInt getCalories() { return calories; }

  public Recipe(String recipeName, OptionalInt calories) {
    this.recipeName = recipeName;
    this.calories = calories;
  }
}

public final class NumericOptionalUsage {
  public static void main(String[] args) {
    // Creating an OptionalInt:
    OptionalInt optNOC0 = OptionalInt.of(3500);
    OptionalInt optNOC1 = OptionalInt.empty();
    // Creating recipes with optional number of calories:
    Recipe recipe0 = new Recipe("Mahi-mahi", optNOC0);
    Recipe recipe1 = new Recipe("Loco moco", optNOC1);

    // Querying an Optional:
    //  System.out.println(recipe1.getCalories()
    //                          .getAsInt());            // NoSuchElementException
    System.out.println((recipe1.getCalories().isPresent()
        ? recipe1.getCalories().getAsInt()
            : "Unknown calories."));       // Unknown calories.

    recipe0.getCalories().ifPresent(s -> System.out.println(s + " calories."));
    System.out.println(recipe0.getCalories().orElse(0) + " calories.");
    System.out.println(recipe1.getCalories().orElseGet(() -> 0) + " calories.");
    //  int noc = recipe1.getCalories()                    // RuntimeException
    //              .orElseThrow(() -> new RuntimeException("Unknown calories."));
  }
}
```

```
void forEach(Consumer<? super T> action)
void forEachOrdered(Consumer<? super T> action)
```

```
Consumer<String> printStr = str -> System.out.print(str + "|");

CD.cdList.stream().map(CD::title).forEach(printStr);               // (1a)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
CD.cdList.stream().parallel().map(CD::title).forEach(printStr);    // (1b)
//Lambda Dancing|Hot Generics|Keep on Erasing|Java Jam|Java Jive|
CD.cdList.stream().map(CD::title).forEachOrdered(printStr);              // (2a)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|

CD.cdList.stream().parallel().map(CD::title).forEachOrdered(printStr);   // (2b)
//Java Jive|Java Jam|Lambda Dancing|Keep on Erasing|Hot Generics|
IntConsumer printInt = n -> out.print(n + "|");

IntStream.of(2018, 2019, 2020, 2021, 2022).forEach(printInt);            // (3a)
//2018|2019|2020|2021|2022|

IntStream.of(2018, 2019, 2020, 2021, 2022).parallel().forEach(printInt); // (3b)
//2020|2019|2018|2021|2022|
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz);   // (1) true
boolean allJazzCds = CD.cdList.stream().allMatch(CD::isJazz);  // (2) false
boolean noJazzCds = CD.cdList.stream().noneMatch(CD::isJazz);  // (3) false
```

```
Predicate<CD> eq2015 = cd -> cd.year().compareTo(Year.of(2015)) == 0;
Predicate<CD> gt2015 = cd -> cd.year().compareTo(Year.of(2015)) > 0;
```

```
boolean noneEQ2015 = CD.cdList.stream().noneMatch(eq2015);     // (4) true
boolean allGT2015 = CD.cdList.stream().allMatch(gt2015);       // (5) true
boolean noneNotGT2015 = CD.cdList.stream().noneMatch(gt2015.negate()); // (6) true
```

```
IntStream yrStream = IntStream.of(2018, 2019, 2020);
IntPredicate isLeapYear = yr -> Year.of(yr).isLeap();
boolean anyLeapYear = yrStream.anyMatch(isLeapYear);
out.println("Any leap year: " + anyLeapYear);   // true
```

```
import static java.lang.System.out;

import java.util.Arrays;
import java.util.stream.IntStream;

public class IdentityMatrixTest {
  public static void main(String[] args) {
    // Matrices to test:
    int[][] sqMatrix1 = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
    int[][] sqMatrix2 = { {1, 1}, {1, 1} };
    isIdentityMatrixLoops(sqMatrix1);
    isIdentityMatrixLoops(sqMatrix2);
    isIdentityMatrixStreams(sqMatrix1);
    isIdentityMatrixStreams(sqMatrix2);
  }

  private static void isIdentityMatrixLoops(int[][] sqMatrix) {           // (1)
    boolean isCorrectValue = false;
    outerLoop:
    for (int i = 0; i < sqMatrix.length; ++i) {
      for (int j = 0; j < sqMatrix[i].length; ++j) {
        isCorrectValue = j == i ? sqMatrix[i][i] == 1
                                : sqMatrix[i][j] == 0;
        if (!isCorrectValue) break outerLoop;
      }
    }
    out.println(Arrays.deepToString(sqMatrix)
        + (isCorrectValue ? " is ": " is not ") + "an identity matrix.");
  }

  private static void isIdentityMatrixStreams(int[][] sqMatrix) {        // (2)
    boolean isCorrectValue =
        IntStream.range(0, sqMatrix.length)
                 .allMatch(i -> IntStream.range(0, sqMatrix[i].length)   // (3)
                                         .allMatch(j -> j == i           // (4)
                                                   ? sqMatrix[i][i] == 1
                                                   : sqMatrix[i][j] == 0));
    out.println(Arrays.deepToString(sqMatrix)
        + (isCorrectValue ? " is ": " is not ") + "an identity matrix.");
  }
}
```

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] is an identity matrix.
[[1, 1], [1, 1]] is not an identity matrix.
```

```
Optional<CD> firstCD1 = CD.cdList.stream().findFirst();          // (1)
out.println(firstCD1.map(CD::title).orElse("No first CD."));    // (2) Java Jive
```

```
Optional<CD> anyCD2 = CD.cdList.stream().parallel().findAny();  // (3)
out.println(anyCD2.map(CD::title).orElse("No CD."));            // Lambda Dancing
```

```
boolean anyJazzCD = CD.cdList.stream().anyMatch(CD::isJazz);              // (5)
out.println("Any Jazz CD: " + anyJazzCD);   // Any Jazz CD: true

Optional<CD> optJazzCD = CD.cdList.stream().filter(CD::isJazz).findAny(); // (6)
optJazzCD.ifPresent(out::println);          // <Jaav, "Java Jam", 6, 2017, JAZZ>
```

```
IntStream numStream = IntStream.of(50, 55, 65, 70, 75, 77);
OptionalInt intOpt = numStream.filter(n -> n % 7 == 0).findAny();
intOpt.ifPresent(System.out::println);      // 70
```

```
Stream.generate(() -> 1).filter(n -> n == 0).findAny();       // Never terminates.
```

```
long numOfCDS = CD.cdList.stream().count();                        // 5
long numOfJazzCDs = CD.cdList.stream().filter(CD::isJazz).count(); // 3
```

```
IntStream numStream = IntStream.rangeClosed(1, 100);
long divBy7 = numStream.filter(n -> n % 7 == 0).count();            // 14
```

```
Optional<T> min(Comparator<? super T> cmp)
Optional<T> max(Comparator<? super T> cmp)
```

```
Optional<CD> minCD = CD.cdList.stream().min(Comparator.naturalOrder());
minCD.ifPresent(out::println);      // <Funkies, "Lambda Dancing", 10, 2018, POP>
out.println(minCD.map(CD::artist).orElse("No min CD."));    // Funkies

Optional<CD> maxCD = CD.cdList.stream().max(Comparator.naturalOrder());
maxCD.ifPresent(out::println);      // <Jaav, "Java Jive", 8, 2017, POP>
out.println(maxCD.map(CD::artist).orElse("No max CD."));    // Jaav
```

```
IntStream iStream = IntStream.rangeClosed(1, 100);
OptionalInt maxNum = iStream.filter(n -> n % 7 == 0).max(); // 98
```

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

$$e_i \oplus e_j \oplus e_k \oplus e_l == (e_i \oplus e_j) \oplus (e_k \oplus e_l)$$

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

```
<U> U reduce(
        U identity,
        BiFunction<U,? super T,U> accumulator,
        BinaryOperator<U> combiner)
```

```
int sum = 0;                                // (1) Initialize the partial result.
for (CD cd : CD.cdList) {                   // (2) Iterate over the list.
  int numOfTracks = cd.noOfTracks();        // (3) Get the current value.
  sum = sum + numOfTracks;                  // (4) Calculate new partial result.
}
```

```
reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks))
reduce(0, Integer::sum)               // Method reference
sum()                                 // Special functional reduction, p. 1001.
```

```
OptionalInt optSumTracks0 = CD.cdList                           // (13)
    .stream()
    .mapToInt(CD::noOfTracks)
    .reduce(Integer::sum);                                      // (14)
out.println("Total number of tracks: " + optSumTracks0.orElse(0));  // 42
```

```
import static java.lang.System.out;

import java.util.Comparator;
import java.util.Optional;
import java.util.OptionalInt;
import java.util.function.BinaryOperator;

public final class FunctionalReductions {
  public static void main(String[] args) {

// Two-argument reduce() method:
  {
    out.println("(1) Find total number of tracks (loop-based version):");
    int sum = 0;                          // (1) Initialize the partial result.
    for (CD cd : CD.cdList) {             // (2) Iterate over the list.
      int numOfTracks = cd.noOfTracks();  // (3) Get the next value.
      sum = sum + numOfTracks;            // (4) Calculate new partial result.
    }
    out.println("Total number of tracks: " + sum);
  }

    out.println("(2) Find total number of tracks (stream-based version):");
    int totNumOfTracks = CD.cdList                        // (5)
        .stream()                                         // (6)
        .mapToInt(CD::noOfTracks)                         // (7)
        .reduce(0,                                        // (8)
                (sum, numOfTracks) -> sum + numOfTracks); // (9)
    //  .reduce(0, (sum, noOfTracks) -> Integer.sum(sum, noOfTracks));
    //  .reduce(0, Integer::sum);
    //  .sum();
    out.println("Total number of tracks: " + totNumOfTracks);
    out.println();

    out.println("(3) Find total number of tracks (accumulator logging): ");
    int totNumOfTracks1 = CD.cdList                       // (10)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce(0,                                        // (11)
            (sum, noOfTracks) -> {                        // (12)
                int newSum = sum + noOfTracks;
                out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
                            sum, noOfTracks, newSum);
                return newSum;
            }
         );
    out.println("Total number of tracks: " + totNumOfTracks1);
    out.println();

// One-argument reduce() method:

    out.println("(4) Find total number of tracks (stream-based version):");
    OptionalInt optSumTracks0 = CD.cdList                 // (13)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce(Integer::sum);                            // (14)
    out.println("Total number of tracks: " + optSumTracks0.orElse(0));
    out.println();

    out.println("(5) Find total number of tracks (accumulator logging): ");
    OptionalInt optSumTracks1 = CD.cdList                 // (15)
        .stream()
        .mapToInt(CD::noOfTracks)
        .reduce((sum, noOfTracks) -> {                    // (16)
           int newSum = sum + noOfTracks;
           out.printf("Accumulator: sum=%2d, noOfTracks=%2d, newSum=%2d%n",
                       sum, noOfTracks, newSum);
           return newSum;
         });
    out.println("Total number of tracks: " + optSumTracks1.orElse(0));
    out.println();

// Three-argument reduce() method:

    out.println("(6) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks5 = CD.cdList                          // (17)
    //  .stream()                                           // (18a)
        .parallelStream()                                   // (18b)
        .reduce(Integer.valueOf(0),                         // (19) Initial value
                (sum, cd) -> sum + cd.noOfTracks(),         // (20) Accumulator
                (sum1, sum2) -> sum1 + sum2);               // (21) Combiner
    out.println("Total number of tracks: " + sumTracks5);
    out.println();

    out.println("(7) Find total number of tracks (accumulator + combiner): ");
    Integer sumTracks6 = CD.cdList                          // (22)
//      .stream()                                           // (23a)
        .parallelStream()                                   // (23b)
        .reduce(0,
               (sum, cd) -> {                               // (24) Accumulator
                 Integer noOfTracks = cd.noOfTracks();
                 Integer newSum = sum + noOfTracks;
                 out.printf("Accumulator: sum=%2d, noOfTracks=%2d, "
                            + "newSum=%2d%n", sum, noOfTracks, newSum);
                 return newSum;
               },
               (sum1, sum2) -> {                            // (25) Combiner
                 Integer newSum = sum1 + sum2;
                 out.printf("Combiner: sum1=%2d, sum2=%2d, newSum=%2d%n",
                            sum1, sum2, newSum);
                 return newSum;
               }
         );
    out.println("Total number of tracks: " + sumTracks6);
    out.println();

    // Compare by CD title.
    Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);    // (26)
    BinaryOperator<CD> maxByTitle =
        (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

    // Query: Find maximum Jazz CD by title:
    Optional<CD> optMaxJazzCD = CD.cdList                   // (28)
        .stream()
        .filter(CD::isJazz)
        .reduce(BinaryOperator.maxBy(cmpByTitle));          // (29a)
    //  .reduce(maxByTitle);                                // (29b)
    //  .max(cmpByTitle);                                   // (29c)
    optMaxJazzCD.map(CD::title).ifPresent(out::println);// Keep on Erasing
  }
}
```

```
(1) Find total number of tracks (loop-based version):
Total number of tracks: 42
(2) Find total number of tracks (stream-based version):
Total number of tracks: 42

(3) Find total number of tracks (accumulator logging):
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(4) Find total number of tracks (stream-based version):
Total number of tracks: 42

(5) Find total number of tracks (accumulator logging):
Accumulator: sum= 8, noOfTracks= 6, newSum=14
Accumulator: sum=14, noOfTracks=10, newSum=24
Accumulator: sum=24, noOfTracks= 8, newSum=32
Accumulator: sum=32, noOfTracks=10, newSum=42
Total number of tracks: 42

(6) Find total number of tracks (accumulator + combiner):
Total number of tracks: 42

(7) Find total number of tracks (accumulator + combiner):
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks=10, newSum=10
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2=10, newSum=18
Combiner: sum1=10, sum2=18, newSum=28
Accumulator: sum= 0, noOfTracks= 6, newSum= 6
Accumulator: sum= 0, noOfTracks= 8, newSum= 8
Combiner: sum1= 8, sum2= 6, newSum=14
Combiner: sum1=14, sum2=28, newSum=42
Total number of tracks: 42

Keep on Erasing
```

```
Integer sumTracks5 = CD.cdList                                  // (17)
    .stream()                                                    // (18a)
//  .parallelStream()                                            // (18b)
    .reduce(Integer.valueOf(0),                                  // (19) Initial value
            (sum, cd) -> sum + cd.noOfTracks(),                  // (20) Accumulator
            (sum1, sum2) -> sum1 + sum2);                        // (21) Combiner
```

```
Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);    // (26)
BinaryOperator<CD> maxByTitle =
    (cd1, cd2) -> cmpByTitle.compare(cd1, cd2) > 0 ? cd1 : cd2; // (27)

Optional<CD> optMaxJazzCD = CD.cdList                          // (28)
    .stream()
    .filter(CD::isJazz)
    .reduce(BinaryOperator.maxBy(cmpByTitle));                 // (29a)
//  .reduce(maxByTitle);                                       // (29b)
//  .max(cmpByTitle);                                          // (29c)
optMaxJazzCD.map(CD::title).ifPresent(out::println);   // Keep on Erasing
```

```
List<String> titles = CD.cdList.stream()
                          .map(CD::title).collect(Collectors.toList());
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

```
<R> R collect(
          Supplier<R>                  supplier,
          BiConsumer<R,? super T> accumulator,
          BiConsumer<R,R>            combiner)
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Stream;

public final class Collecting {
  public static void main(String[] args) {

    // Query: Create a list with the number of tracks on each CD.
    System.out.println("Sequential Mutable Reduction:");
    List<Integer> tracks = CD.cdList                              // (1)
        .stream()                                                 // (2a)
//      .parallelStream()                                         // (2b)
        .map(CD::noOfTracks)                                      // (3)
        .collect(() -> new ArrayList<>(),                         // (4) Supplier
                 (cont, noOfTracks) -> cont.add(noOfTracks),      // (5) Accumulator
                 (cont1, cont2) -> cont1.addAll(cont2));          // (6) Combiner
//      .collect(ArrayList::new, ArrayList::add, ArrayList::addAll); // (6a)
//      .toList();
    System.out.println("Number of tracks on each CD (sequential): " + tracks);
    System.out.println();

    System.out.println("Parallel Mutable Reduction:");
    List<Integer> tracks1 = CD.cdList                             // (7)
//      .stream()                                              // (8a)
        .parallelStream()                                      // (8b)
        .map(CD::noOfTracks)                                   // (9)
        .collect(                                              // (10)
            () -> {                                            // (11) Supplier
              System.out.println("Supplier: Creating an ArrayList");
              return new ArrayList<>();
            },
            (cont, noOfTracks) -> {                            // (12) Accumulator
              System.out.printf("Accumulator: cont:%s, noOfTracks:%s",
                                   cont, noOfTracks);
              cont.add(noOfTracks);
              System.out.printf(", mutCont:%s%n", cont);
            },
            (cont1, cont2) -> {                                // (13) Combiner
              System.out.printf("Combiner: con1:%s, cont2:%s", cont1, cont2);
              cont1.addAll(cont2);
              System.out.printf(", mutCont:%s%n", cont1);
            });
    System.out.println("Number of tracks on each CD (parallel): " + tracks1);
    System.out.println();

    // Query: Create an ordered set with CD titles, according to natural order.
    Set<String> cdTitles = CD.cdList                           // (14)
        .stream()
        .map(CD::title)
        .collect(TreeSet::new, TreeSet::add, TreeSet::addAll);// (15)
    System.out.println("CD titles: " + cdTitles);
    System.out.println();

    // Query: Go bananas.
    StringBuilder goneBananas = Stream                         // (16)
        .iterate("ba", b -> b + "na")                          // (17)
        .limit(5)
        .peek(System.out::println)
        .collect(StringBuilder::new,                           // (18)
                 StringBuilder::append,
                 StringBuilder::append);
    System.out.println("Go bananas: " + goneBananas);
  }
}
```

```
Sequential Mutable Reduction:
Number of tracks on each CD (sequential): [8, 6, 10, 8, 10]

Parallel Mutable Reduction:
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:6, mutCont:[6]
Combiner: con1:[8], cont2:[6], mutCont:[8, 6]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Supplier: Creating an ArrayList
```

```
Accumulator: cont:[], noOfTracks:8, mutCont:[8]
Combiner: con1:[10], cont2:[8], mutCont:[10, 8]
Supplier: Creating an ArrayList
Accumulator: cont:[], noOfTracks:10, mutCont:[10]
Combiner: con1:[10, 8], cont2:[10], mutCont:[10, 8, 10]
Combiner: con1:[8, 6], cont2:[10, 8, 10], mutCont:[8, 6, 10, 8, 10]
Number of tracks on each CD (parallel): [8, 6, 10, 8, 10]

CD titles: [Hot Generics, Java Jam, Java Jive, Keep on Erasing, Lambda Dancing]

ba
bana
banana
bananana
banananana
Go bananas: babanabananabanananabanananana
```

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
Object[] objArray = CD.cdList.stream().map(CD::title)
                                      .toArray();                     // (1)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
String[] cdTitles = CD.cdList.stream().map(CD::title)
                                      .toArray(String[]::new);         // (2)
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
int[] intArray1 = IntStream.iterate(1, i -> i + 1).limit(5).toArray();// (3)
// [1, 2, 3, 4, 5]
int[] intArray2 = IntStream.range(-5, 5).toArray();                    // (4)
// [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
int[] intArray3 = IntStream.iterate(1, i -> i + 1)                  // (5)
                           .toArray();                        // OutOfMemoryError!
```

```
CD[] cdArray1 = CD.cdList.stream().toArray(CD[]::new);       // (6) Preferred.
CD[] cdArray2 = CD.cdList.toArray(new CD[CD.cdList.size()]); // (7) Not efficient.
```

```
List<String> titles = CD.cdList.stream().map(CD::title).toList();
// [Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles");          // UnsupportedOperationException!
```

NumType`SummaryStatistics summaryStatistics()`

```
int totNumOfTracks = CD.cdList
    .stream()                                          // Stream<CD>
    .mapToInt(CD::noOfTracks)                          // IntStream
    .sum();                                            // 42
```

```
int sumEven = IntStream
    .rangeClosed(1, 100)
    .filter(i -> i % 2 == 0)
    .sum();                                            // 2550
```

```
int numOfCDs = CD.cdList
    .stream()
    .mapToInt(cd -> 1)                                    // CD => 1
    .sum();                                               // 5
```

```
double total = DoubleStream.empty().sum();         // 0.0
```

```
OptionalDouble optAverage = CD.cdList
    .stream()
    .mapToInt(CD::noOfTracks)
    .average();
System.out.println(optAverage.orElse(0.0));        // 8.4
```

```
void combine(NumTypeSummaryStatistics other)
```

```
IntSummaryStatistics stats1 = List.of(CD.cd0, CD.cd1)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println("Count="   + stats1.getCount());       // Count=2
System.out.println("Sum="     + stats1.getSum());         // Sum=14
System.out.println("Min="     + stats1.getMin());         // Min=6
System.out.println("Max="     + stats1.getMax());         // Max=8
System.out.println("Average=" + stats1.getAverage());     // Average=7.0
System.out.println(stats1);
//IntSummaryStatistics{count=2, sum=14, min=6, average=7.000000, max=8}
```

```
stats1.accept(CD.cd2.noOfTracks());      // Add the value 10.
System.out.println(stats1);
//IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

```
IntSummaryStatistics stats2 = List.of(CD.cd3, CD.cd4)
    .stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();
System.out.println(stats2);
//IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10}
```

```
stats1.combine(stats2);                    // Combine stats2 with stats1.
System.out.println(stats1);
//IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

```
IntSummaryStatistics emptyStats = IntStream.empty().summaryStatistics();
System.out.println(emptyStats);
//IntSummaryStatistics{count=0, sum=0, min=2147483647, average=0.000000,
//max=-2147483648}
```

```
import static java.util.stream.Collectors.*;
import java.util.stream.Collectors;
```

```
ArrayList<String> cdTitles1 = CD.cdList.stream() // Stream<CD>
    .map(CD::title)                              // Stream<String>
    .collect(Collectors.toCollection(ArrayList::new));
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
static <T,C extends Collection<T>> Collector<T,?,C>
       toCollection(Supplier<C> collectionFactory)
```

```
static <T> Collector<T,?,List<T>> toList()
static <T> Collector<T,?,List<T>> toUnmodifiableList()
static <T> Collector<T,?,Set<T>> toSet()
static <T> Collector<T,?,Set<T>> toUnmodifiableSet()
```

```
List<String> cdTitles3 = CD.cdList.stream()      // Stream<CD>
    .map(CD::title)                              // Stream<String>
    .collect(Collectors.toList());
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
titles.add("Java Jingles");                      // OK
```

```
Set<String> cdTitles2 = CD.cdList.stream()       // Stream<CD>
    .map(CD::title)                              // Stream<String>
    .collect(Collectors.toSet());
//[Hot Generics, Java Jive, Lambda Dancing, Keep on Erasing, Java Jam]
```

```
static <T,K,U> Collector<T,?,Map<K,U>> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper)

static <T,K,U> Collector<T,?,Map<K,U>> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper,
       BinaryOperator<U>                mergeFunction)

static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(
       Function<? super T,? extends K> keyMapper,
       Function<? super T,? extends U> valueMapper,
       BinaryOperator<U>                mergeFunction,
       Supplier<M>                      mapSupplier)
```

```
Map<CD, Year> mapCDToYear = CD.cdList.stream()
    .collect(Collectors.toMap(Function.identity(), CD::year)); // Map<CD, Year>
```

```
List<CD> dupList = List.of(CD.cd0, CD.cd1, CD.cd2, CD.cd0, CD.cd1);
Map<String, Year> mapTitleToYear1 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year));       // (1)
// IllegalStateException: Duplicate key 2017
```

```
Map<String, Year> mapTitleToYear2 = dupList.stream()
    .collect(Collectors.toMap(CD::title, CD::year, (y1, y2) -> y1));       // (1)
```

```
Map<Year, String> mapTitleToYear3 = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
                              (tt, t) -> tt + ":" + t));
//{2017=Java Jive:Java Jam, 2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

```
TreeMap<Year, String> mapYearToLongestTitle = CD.cdList.stream()
    .collect(Collectors.toMap(CD::year, CD::title,
                              BinaryOperator.maxBy(Comparator.naturalOrder()),
                              TreeMap::new));
//{2017=Java Jive, 2018=Lambda Dancing}
```

```
(str1, str2) -> str1.compareTo(str2) > 0 ? str1 : str2
```

```
ConcurrentMap<Year, String> concMapYearToTitles = CD.cdList
    .parallelStream()
    .collect(Collectors.toConcurrentMap(CD::year, CD::title,
                                        (tt, t) -> tt + ":" + t));
//{2017=Java Jam:Java Jive, 2018=Lambda Dancing:Hot Generics:Keep on Erasing}
```

```
static Collector<CharSequence,?,String> joining()
static Collector<CharSequence,?,String> joining(CharSequence delimiter)
static Collector<CharSequence,?,String> joining(CharSequence delimiter,
                                                CharSequence prefix,
                                                CharSequence suffix)
```

```
String concatTitles1 = CD.cdList.stream()          // Stream<CD>
    .map(CD::title)                                // Stream<String>
    .collect(Collectors.joining());                // (1)
//Java JiveJava JamLambda DancingKeep on ErasingHot Generics
```

```
String concatTitles2 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", "));           // (2) Delimiter
//Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics
```

```
String concatTitles3 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.joining(", ", "[", "]"));  // (3) Delimiter, Prefix, Suffix
//[Java Jive, Java Jam, Lambda Dancing, Keep on Erasing, Hot Generics]
```

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
       Function<? super T,? extends K> classifier)

static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
       Function<? super T,? extends K> classifier,
       Collector<? super T,A,D>        downstream)

static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(
       Function<? super T,? extends K> classifier,
       Supplier<M>                     mapSupplier,
       Collector<? super T,A,D>        downstream)
```

```
Map<Integer, List<CD>> map22 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList()));  // (2)
```

```
Map<Integer, List<CD>> map33 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,                         // (3)
                                   TreeMap::new,
                                   Collectors.toList()));
```

```
Map<Integer, Set<CD>> map44 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toSet()));   // (4)
```

```
{
6=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
8=[<Jaav, "Java Jive", 8, 2017, POP>,
   <Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
10=[<Funkies, "Lambda Dancing", 10, 2018, POP>,
    <Genericos, "Hot Generics", 10, 2018, JAZZ>]
}
```

```
Map<Integer, Long> map55 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.counting()));
//{6=1, 8=2, 10=2}
```

```
Map<Integer, Map<Genre, List<CD>>> twoLevelGrp = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,              // (1)
                Collectors.groupingBy(CD::genre)));            // (2)
```

```
{
6={JAZZ=[<Jaav, "Java Jam", 6, 2017, JAZZ>]},
8={JAZZ=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>],
   POP=[<Jaav, "Java Jive", 8, 2017, POP>]},
10={JAZZ=[<Genericos, "Hot Generics", 10, 2018, JAZZ>],
    POP=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
}
```

```
Map<Integer, Map<Genre, Long>> twoLevelGrp2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,
                Collectors.groupingBy(CD::genre,
                                     Collectors.counting())));   // (3)
```

```
{6={JAZZ=1}, 8={JAZZ=1, POP=1}, 10={JAZZ=1, POP=1}}
```

```
ConcurrentMap<Integer, Long> map66 = CD.cdList
    .parallelStream()
    .collect(Collectors.groupingByConcurrent(CD::noOfTracks,
                                             Collectors.counting()));
//{6=1, 8=2, 10=2}
```

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
       Predicate<? super T>     predicate)

static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
       Predicate<? super T>     predicate,
       Collector<? super T,A,D> downstream)
```

```
{false=[<Jaav, "Java Jam", 6, 2017, JAZZ>,
        <Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Generics", 10, 2018, JAZZ>],
 true=[<Jaav, "Java Jive", 8, 2017, POP>,
       <Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

```
List<CD> popCDs = map1.get(true);
List<CD> nonPopCDs = map1.get(false);
```

```
Map<Boolean, List<CD>> map2 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList())); // (2)
```

```
Map<Boolean, Long> map3 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.counting()));
//{false=3, true=2}
```

```
Map<Boolean, Map<Year, List<CD>>> map1 = CD.cdList.stream()
    .collect(Collectors.partitioningBy(CD::isPop,                         // (1)
                 Collectors.groupingBy(CD::year)));                       // (2)
```

```
{false={2017=[<Jaav, "Java Jam", 6, 2017, JAZZ>],
        2018=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
              <Genericos, "Hot Generics", 10, 2018, JAZZ>]},
 true={2017=[<Jaav, "Java Jive", 8, 2017, POP>],
       2018=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}}
```

```
static <T,A,R> Collector<T,?,R> filtering(
       Predicate<? super T>     predicate,
       Collector<? super T,A,R> downstream)
```

```
// Filtering downstream from grouping.
Map<Integer, List<CD>> grpByTracksFilterByPopCD = CD.cdList.stream()
    .collect(Collectors.groupingBy(CD::noOfTracks,                      // (1)
                Collectors.filtering(CD::isPop, Collectors.toList()))); // (2)
```

```
{6=[],
 8=[<Jaav, "Java Jive", 8, 2017, POP>],
 10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

```
// Filtering before grouping.
Map<Integer, List<CD>> filterByPopCDGrpByTracks = CD.cdList.stream()
    .filter(CD::isPop)                                              // (1)
    .collect(Collectors.groupingBy(CD::noOfTracks, Collectors.toList()));
```

```
{8=[<Jaav, "Java Jive", 8, 2017, POP>],
 10=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

```
// Filtering downstream from partitioning.
Map<Boolean, List<CD>> partbyPopCDsFilterByYear = CD.cdList.stream()     // (1)
    .collect(Collectors.partitioningBy(CD::isPop,
                Collectors.filtering(cd -> cd.year().equals(Year.of(2018)),
                                     Collectors.toList()))); // (2)

// Filtering before partitioning.
Map<Boolean, List<CD>> filterByYearPartbyPopCDs = CD.cdList.stream()     // (2)
    .filter(cd -> cd.year().equals(Year.of(2018)))
    .collect(Collectors.partitioningBy(CD::isPop, Collectors.toList()));
```

```
{false=[<Genericos, "Keep on Erasing", 8, 2018, JAZZ>,
        <Genericos, "Hot Generics", 10, 2018, JAZZ>],
 true=[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
```

```
static <T,U,A,R> Collector<T,?,R> mapping(
       Function<? super T,? extends U> mapper,
       Collector<? super U,A,R>        downstream)
```

```
Map<Year, Set<String>> titlesByYearInSet = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                              // (1)
            CD::title,                                   // Mapper
            Collectors.toSet())));                       // Downstream collector
System.out.println(titlesByYearInSet);
// {2017=[Java Jive, Java Jam],
//  2018=[Hot Generics, Lambda Dancing, Keep on Erasing]}
```

```
Map<Year, String> joinTitlesByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::year,
        Collectors.mapping(                               // (2)
            CD::title,
            Collectors.joining(":"))));
System.out.println(joinTitlesByYear);
// {2017=Java Jive:Java Jam,
//  2018=Lambda Dancing:Keep on Erasing:Hot Generics}
```

```
Map<Year, Long> TotalNumOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
       CD::year,
       Collectors.mapping(                                  // (3)
           CD::noOfTracks,
           Collectors.counting())));
System.out.println(TotalNumOfTracksByYear);                 // {2017=2, 2018=3}
```

```
static <T,U,A,R> Collector<T,?,R> flatMapping(
       Function<? super T,? extends Stream<? extends U>> mapper,
       Collector<? super U,A,R>                          downstream)
```

```
// Given lists of CDs:
List<CD> cdListA = List.of(CD.cd0, CD.cd1);
List<CD> cdListB = List.of(CD.cd0, CD.cd1, CD.cd1);

// Find unique CD titles in the given lists:
Set<String> set =
  Stream.of(cdListA, cdListB)                         // (1) Stream<List<CD>>
        .collect(Collectors.flatMapping(List::stream, // (2) Flatten to Stream<CD>
             Collectors.mapping(CD::title,            // (3) Stream<String>
                 Collectors.toSet())));               // (4) Set<String>
```

```
import java.util.List;

// Radio station with a playlist.
public class RadioPlaylist {                                                    // (1)
  private String radioStationName;
  private List<CD> Playlist;

  public RadioPlaylist(String radioStationName, List<CD> cdList) {
    this.radioStationName = radioStationName;
    this.Playlist = cdList;
  }

  public String getRadioStationName() { return this.radioStationName; }
  public List<CD> getPlaylist() { return this.Playlist; }
}
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectorsFlatMapping {
  public static void main(String[] args) {
    // Some lists of CDs:                                                      (2)
    List<CD> cdList1 = List.of(CD.cd0, CD.cd1, CD.cd1, CD.cd2);
    List<CD> cdList2 = List.of(CD.cd0, CD.cd0, CD.cd3);
    List<CD> cdList3 = List.of(CD.cd0, CD.cd4);

    // Some radio playlists:                                                   (3)
    RadioPlaylist pl1 = new RadioPlaylist("Radio JVM", cdList1);
    RadioPlaylist pl2 = new RadioPlaylist("Radio JRE", cdList2);
    RadioPlaylist pl3 = new RadioPlaylist("Radio JAR", cdList3);

    // List of radio playlists:                                                (4)
    List<RadioPlaylist> radioPlaylists = List.of(pl1, pl2, pl3);

    // Map of radio station names and set of CD titles they played:            (5)
    Map<String, Set<String>> map = radioPlaylists.stream()                  // (6)
        .collect(Collectors.groupingBy(RadioPlaylist::getRadioStationName, // (7)
             Collectors.flatMapping(rpl -> rpl.getPlaylist().stream(),     // (8)
                 Collectors.mapping(CD::title,                             // (9)
                     Collectors.toSet()))));                               // (10)
    System.out.println(map);
  }
}
```

```
{Radio JAR=[Hot Generics, Java Jive],
 Radio JVM=[Java Jive, Lambda Dancing, Java Jam],
 Radio JRE=[Java Jive, Keep on Erasing]}
```

```
static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(
       Collector<T,A,R> downstream,
       Function<R,RR>   finisher)
```

```
Map<Year, Integer> maxTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.collectingAndThen(                                   // (1)
             Collectors.maxBy(Comparator.comparing(CD::noOfTracks)),    // (2)
             optCD -> optCD.map(CD::noOfTracks).orElse(0)))             // (3)
     );
System.out.println(maxTracksByYear);                       // {2017=8, 2018=10}
```

```
Map<Genre, String> avgTracksByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::genre,
         Collectors.collectingAndThen(                                  // (4)
             Collectors.averagingDouble(CD::noOfTracks),                // (5)
             d -> String.format("%.1f", d)))                            // (6)
     );
System.out.println(avgTracksByGenre);                    // {JAZZ=8.0, POP=9.0}
```

```
static <T> Collector<T,?,Long> counting()
```

```
Long numOfJazzCds1 = CD.cdList.stream().filter(CD::isJazz)
    .collect(Collectors.counting());                  // (1) Standalone collector
System.out.println(numOfJazzCds1);                    // 3
```

```
Map<Genre, Long> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
                 CD::genre,
                 Collectors.counting()));            // (2) Downstream collector
System.out.println(grpByGenre);                      // {POP=2, JAZZ=3}
System.out.println(grpByGenre.get(Genre.JAZZ));      // 3
```

```
long numOfJazzCds2 = CD.cdList.stream().filter(CD::isJazz)
    .count();                                          // (3) Stream.count()
System.out.println(numOfJazzCds2);                     // 3
```

```
static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> cmp)
static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> cmp)
```

```
Comparator<CD> natCmp = Comparator.naturalOrder(); // (1)

Optional<CD> maxCD = CD.cdList.stream()
    .collect(Collectors.maxBy(natCmp));            // (2) Standalone collector
System.out.println("Max CD: "
    + maxCD.map(CD::title).orElse("No CD."));      // Max CD: Java Jive
```

```
// Group CDs by musical genre, and max CD in each group.
Map<Genre, Optional<CD>> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
        CD::genre,
        Collectors.maxBy(natCmp)));       // (3) Downstream collector
System.out.println(grpByGenre);
//{JAZZ=Optional[<Jaav, "Java Jam", 6, 2017, JAZZ>],
// POP=Optional[<Jaav, "Java Jive", 8, 2017, POP>]}

System.out.println("Title of max Jazz CD: "
    + grpByGenre.get(Genre.JAZZ)
                .map(CD::title)
                .orElse("No CD."));       // Title of max Jazz CD: Java Jam
```

```
Optional<CD> maxCD1 = CD.cdList.stream()
    .max(natCmp);                          // (4) max() on Stream<CD>.
System.out.println("Title of max CD: "
    + maxCD1.map(CD::title)
            .orElse("No CD."));            // Title of max CD: Java Jive
```

```
static <T> Collector<T,?,NumType> summingNumType(
        ToNumTypeFunction<? super T> mapper)
```

```
Integer sumTracks = CD.cdList.stream()
    .collect(Collectors.summingInt(CD::noOfTracks));   // (1) Standalone collector
System.out.println(sumTracks);                         // 42
```

```
Map<Genre, Integer> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::genre,
         Collectors.summingInt(CD::noOfTracks)));    // (2) Downstream collector
System.out.println(grpByGenre);                      // {POP=18, JAZZ=24}
System.out.println(grpByGenre.get(Genre.JAZZ));      // 24
```

```
int sumTracks2 = CD.cdList.stream()                    // (3) Stream<CD>
    .mapToInt(CD::noOfTracks)                          // IntStream
    .sum();
System.out.println(sumTracks2);                        // 42
```

```
static <T> Collector<T,?,Double> averagingNumType(
         ToNumTypeFunction<? super T> mapper)
```

```
Double avgNoOfTracks1 = CD.cdList.stream()
    .collect(Collectors
        .averagingInt(CD::noOfTracks));              // (1) Standalone collector
System.out.println(avgNoOfTracks1);                  // 8.4
```

```
Map<Genre, Double> grpByGenre = CD.cdList.stream()
    .collect(Collectors.groupingBy(
       CD::genre,
       Collectors.averagingInt(CD::noOfTracks)      // (2) Downstream collector
       ));
System.out.println(grpByGenre);                     // {POP=9.0, JAZZ=8.0}
System.out.println(grpByGenre.get(Genre.JAZZ));     // 8.0
```

```
OptionalDouble avgNoOfTracks2 = CD.cdList.stream()  // Stream<CD>
    .mapToInt(CD::noOfTracks)                        // IntStream
    .average();                                      // (3)
System.out.println(avgNoOfTracks2.orElse(0.0));     // 8.4
```

```
static <T> Collector<T,?,NumTypeSummaryStatistics> summarizingNumType(
       ToNumTypeFunction<? super T> mapper)
```

```
IntSummaryStatistics stats1 = CD.cdList.stream()
    .collect(
      Collectors.summarizingInt(CD::noOfTracks)      // (1) Standalone collector
     );
```

```
System.out.println(stats1);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

```
Map<Genre, IntSummaryStatistics> grpByGenre = CD.cdList.stream()
  .collect(Collectors.groupingBy(
     CD::genre,
     Collectors.summarizingInt(CD::noOfTracks)));    // (2) Downstream collector
System.out.println(grpByGenre);
//{POP=IntSummaryStatistics{count=2, sum=18, min=8, average=9.000000, max=10},
// JAZZ=IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}}

System.out.println(grpByGenre.get(Genre.JAZZ));   // Summary stats for Jazz CDs.
// IntSummaryStatistics{count=3, sum=24, min=6, average=8.000000, max=10}
```

```
IntSummaryStatistics stats2 = CD.cdList.stream()
    .mapToInt(CD::noOfTracks)
    .summaryStatistics();                              // (3)
System.out.println(stats2);
// IntSummaryStatistics{count=5, sum=42, min=6, average=8.400000, max=10}
```

```
static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> bop)
static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> bop)
static <T,U> Collector<T,?,U> reducing(
       U                                 identity,
       Function<? super T,? extends U> mapper,
       BinaryOperator<U>                 bop)
```

```
// Comparator to compare CDs by title.
Comparator<CD> cmpByTitle = Comparator.comparing(CD::title);        // (1)
// Comparator to compare strings by their length.
Comparator<String> byLength = Comparator.comparing(String::length); // (2)
```

```
Optional<String> longestTitle1 = CD.cdList.stream()
    .map(CD::title)
    .collect(Collectors.reducing(
        BinaryOperator.maxBy(byLength)));             // (3) Standalone collector
System.out.println(longestTitle1.orElse("No title"));// Keep on Erasing

Optional<String> longestTitle2 = CD.cdList.stream()  // Stream<CD>
    .map(CD::title)                                   // Stream<String>
    .reduce(BinaryOperator.maxBy(byLength));         // (4) Stream.reduce(bop)
```

```
Map<Year, Optional<CD>> cdWithMaxTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
          Collectors.reducing(                          // (5) Downstream collector
              BinaryOperator.maxBy(cmpByTitle))
          ));
System.out.println(cdWithMaxTitleByYear);
// {2017=Optional[<Jaav, "Java Jive", 8, 2017, POP>],
//  2018=Optional[<Funkies, "Lambda Dancing", 10, 2018, POP>]}
System.out.println(cdWithMaxTitleByYear.get(Year.of(2018))
                       .map(CD::title).orElse("No title")); // Lambda Dancing
```

```
Map<Year, String> longestTitleByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.reducing("", CD::title,          // (6) Downstream collector
             BinaryOperator.maxBy(byLength))
         ));
System.out.println(longestTitleByYear);  // {2017=Java Jive, 2018=Keep on Erasing}
System.out.println(longestTitleByYear.get(Year.of(2018)));      // Keep on Erasing

Map<Year, Optional<String>> longestTitleByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.mapping(CD::title,               // (7) Downstream collector
             Collectors.maxBy(byLength))
         ));
System.out.println(longestTitleByYear2);
// {2017=Optional[Java Jive], 2018=Optional[Keep on Erasing]}
System.out.println(longestTitleByYear2.get(Year.of(2018))
                       .orElse("No title."));        // Keep on Erasing
```

```
Map<Year, Integer> noOfTracksByYear = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.reducing(                        // (8) Downstream collector
             0, CD::noOfTracks, Integer::sum)));
System.out.println(noOfTracksByYear);                   // {2017=14, 2018=28}
System.out.println(noOfTracksByYear.get(Year.of(2018)));// 28
```

```
Map<Year, Integer> noOfTracksByYear2 = CD.cdList.stream()
    .collect(Collectors.groupingBy(
         CD::year,
         Collectors.summingInt(CD::noOfTracks)));    // (9) Special case collector
```

```
return LongStream.rangeClosed(1L, n).sum();                // Sequential stream
...
return LongStream.rangeClosed(1L, n).parallel().sum();     // Parallel stream
return LongStream.iterate(1L, i -> i + 1).limit(n).sum();            // Sequential
...
return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum(); // Parallel
```

```
public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
  // ...
  double start = System.nanoTime();                                 // (8)
  result = func.apply(n);                                           // (9)
  double duration = (System.nanoTime() - start)/1_000_000;          // (10) ms.
  // ...
}
```

```
import java.util.function.LongFunction;
import java.util.stream.LongStream;
/*
 * Benchmark the execution time to sum numbers from 1 to n values
 * using streams.
 */
public final class StreamBenchmarks {

  public static long seqSumRangeClosed(long n) {                       // (1)
    return LongStream.rangeClosed(1L, n).sum();
  }

  public static long paraSumRangeClosed(long n) {                      // (2)
    return LongStream.rangeClosed(1L, n).parallel().sum();
  }

  public static long seqSumIterate(long n) {                           // (3)
    return LongStream.iterate(1L, i -> i + 1).limit(n).sum();
  }

  public static long paraSumIterate(long n) {                          // (5)
    return LongStream.iterate(1L, i -> i + 1).limit(n).parallel().sum();
  }

  public static long iterSumLoop(long n) {                             // (5)
    long result = 0;
    for (long i = 1L; i <= n; i++) {
      result += i;
    }
    return result;
  }
  /*
   * Applies the function parameter func, passing n as parameter.
   * Returns the average time (ms.) to execute the function 100 times.
   */
  public static <R> double measurePerf(LongFunction<R> func, long n) { // (6)
    int numOfExecutions = 100;
    double totTime = 0.0;
    R result = null;
    for (int i = 0; i < numOfExecutions; i++) {                        // (7)
      double start = System.nanoTime();                                // (8)
      result = func.apply(n);                                          // (9)
      double duration = (System.nanoTime() - start)/1_000_000;         // (10)
      totTime += duration;                                             // (11)
    }
    double avgTime = totTime/numOfExecutions;                          // (12)
    return avgTime;
  }

  /*
   * Executes the functions in the varargs parameter funcs
   * for different stream sizes.
   */
  public static <R> void xqtFunctions(LongFunction<R>... funcs) {      // (13)
    long[] sizes = {1_000L, 10_000L, 100_000L, 1_000_000L};           // (14)

    // For each stream size ...
    for (int i = 0; i < sizes.length; ++i) {                           // (15)
      System.out.printf("%7d", sizes[i]);
      // ... execute the functions passed in the varargs parameter funcs.
      for (int j = 0; j < funcs.length; ++j) {                         // (16)
        System.out.printf("%10.5f", measurePerf(funcs[j], sizes[i]));
      }
      System.out.println();
    }
  }

  public static void main(String[] args) {                             // (17)

    System.out.println("Streams created with the rangeClosed() method:");// (18)
    System.out.println("  Size   Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumRangeClosed,
                 StreamBenchmarks::paraSumRangeClosed);

    System.out.println("Streams created with the iterate() method:");  // (19)
    System.out.println("  Size   Sequential Parallel");
    xqtFunctions(StreamBenchmarks::seqSumIterate,
                 StreamBenchmarks::paraSumIterate);

    System.out.println("Iterative solution with an explicit loop:");   // (20)
    System.out.println("  Size   Iterative");
    xqtFunctions(StreamBenchmarks::iterSumLoop);
  }
}
```

```
Streams created with the rangeClosed() method:
  Size   Sequential Parallel
   1000   0.05681   0.11031
  10000   0.06698   0.13979
 100000   0.71274   0.52627
1000000   7.02237   4.37249
Streams created with the iterate() method:
  Size   Sequential Parallel
   1000   0.08645   0.34696
  10000   0.35687   1.27861
 100000   3.24083  11.38709
1000000  29.92285 117.87909
Iterative solution with an explicit loop:
  Size   Iterative
   1000   0.00586
  10000   0.02330
 100000   0.22352
1000000   2.49677
```

```
import java.util.*;

public class RQ1 {
  public static void main(String[] args) {
    List<String> values = Arrays.asList("X", "XXX", "XX", "XXXX");
    int value = values.stream()
                      .mapToInt(v -> v.length())
                      .filter(v -> v != 4)
                      .reduce(1, (x, y) -> x * y);
    System.out.println(value);
  }
}
```

```
import java.util.stream.*;

public class RQ3 {
  public static void main(String[] args) {
    IntStream values = IntStream.range(0, 5);
    // (1) INSERT CODE HERE
    System.out.println(sum);
  }
}
```

```
import java.util.stream.*;

public class RQ4 {
  public static void main(String[] args) {
    IntStream values = IntStream.range(0, 5);
    // (1) INSERT CODE HERE
    System.out.println(value);
  }
}
```

```
import java.util.*;
import java.util.stream.*;

public class RQ5 {
  public static void main(String[] args) {
    List<String> values = List.of("AA", "BBB", "C", "DD", "EEE");
    Map<Integer, List<String>> map = null;
    // (1) INSERT CODE HERE
    map.forEach((i, s) -> System.out.println(i + " " + s));
  }
}
```

```
map = values.stream()
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.filtering(s -> !s.contains("C"),
                             Collectors.toList())));
```

```
map = values.stream()
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.filtering(s -> s.contains("C"),
                             Collectors.toList())));
```

```
map = values.stream()
            .filter(s -> !s.contains("C"))
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.toList()));
```

```
map = values.stream()
            .filter(s -> s.contains("C"))
            .collect(Collectors.groupingBy(s -> s.length(),
                         Collectors.toList()));
```

```
import java.util.stream.*;

public class RQ7 {
  public static void main(String[] args) {
    Stream<String> values = Stream.generate(() -> "A");
    boolean value = values.peek(v -> System.out.print("B"))
                          .takeWhile(v -> !v.equals("A"))
                          .peek(v -> System.out.print("C"))
                          .anyMatch(v -> v.equals("A"));
    System.out.println(value);
  }
}
```

```
import java.util.stream.*;

public class RQ9 {
  public static void main(String[] args) {
    IntStream.range('a', 'e')
             .mapToObj(i -> String.valueOf((char) i).toUpperCase())
             .filter(s -> "AEIOU".contains(s))
             .forEach(s -> System.out.print(s));
  }
}
```

```
import java.util.stream.*;

public class RQ10 {
  public static void main(String[] args) {
    IntStream.range(0, 5)
             .filter(i -> i % 2 != 0)
             .forEach(i -> System.out.println(i));
  }
}
```

```
IntStream.rangeClosed(0, 5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

```
IntStream.range(0, 10)
         .takeWhile(i -> i < 5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

```
IntStream.range(0, 10)
         .limit(5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

```
IntStream.generate(() -> {int x = 0; return x++;})
         .takeWhile(i -> i < 4)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

```
var x = 0;
IntStream.generate(() -> return x++)
         .limit(5)
         .filter(i -> i % 2 != 0)
         .forEach(i -> System.out.println(i));
```

```
import java.util.function.*;
import java.util.stream.*;

public class RQ11 {
  public static void main(String[] args) {
    Stream<String> abc = Stream.of("A", "B", "C");
    Stream<String> xyz = Stream.of("X", "Y", "Z");
    String value = Stream.concat(xyz, abc).reduce((a, b) -> b + a).get();
    System.out.println(value);
  }
}
```

```
Stream.of("A", "B", "C", "D", "E")
      .filter(s -> s.compareTo("B") < 0)
      .collect(Collectors.groupingBy(s -> "AEIOU".contains(s)))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

```
Stream.of("A", "B", "C", "D", "E")
      .filter(s -> s.compareTo("B") < 0)
      .collect(Collectors.partitioningBy(s -> "AEIOU".contains(s)))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

```
Stream.of("A", "B", "C", "D", "E")
      .collect(Collectors.groupingBy(s -> "AEIOU".contains(s),
                   Collectors.filtering(s -> s.compareTo("B") < 0,
                                        Collectors.toList())))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

```
Stream.of("A", "B", "C", "D", "E")
      .collect(Collectors.partitioningBy(s -> "AEIOU".contains(s),
                   Collectors.filtering(s -> s.compareTo("B") < 0,
                                        Collectors.toList())))
      .forEach((x, y) -> System.out.println(x + " " + y));
```

```
import java.util.stream.*;

public class RQ13 {
  public static void main(String[] args) {
    Stream<String> strings = Stream.of("i", "am", "ok").parallel();
    IntStream chars = strings.flatMapToInt(line -> line.chars()).sorted();
    chars.forEach(c -> System.out.print((char)c));
  }
}
```

```
import java.util.stream.*;

public class RQ17 {
  public static void main(String[] args) {
    Stream<String> values = Stream.of("is", "this", "", null, "ok", "?");
    // (1) INSERT CODE HERE
    System.out.println(c);
  }
}
```

```
Set<String> set1 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                         .filter(v -> v != null)
                         .collect(Collectors.toSet());
```

```
set1.stream()
    .mapToInt(v -> v.length())
    .sorted()
    .forEach(v -> System.out.print(v));
```

```
Set<Integer> set2 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                          .map(v -> (v == null) ? 0 : v.length())
                          .filter(v -> v != 0)
                          .collect(Collectors.toSet());
set2.stream()
    .sorted()
    .forEach(v -> System.out.print(v));
```

```
List<Integer> list1 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                            .map(v -> (v == null) ? 0 : v.length())
                            .filter(v -> v != 0)
                            .toList();
list1.stream()
     .sorted()
     .forEach(v -> System.out.print(v));
```

```
List<Integer> list2 = Stream.of("XX", "XXXX", "", null, "XX", "X")
                            .map(v -> (v == null) ? 0 : v.length())
                            .filter(v -> v != 0)
                            .distinct()
                            .toList();
 list2.stream()
      .sorted()
      .forEach(v -> System.out.print(v));
```

```
// LocalTime
static LocalTime of(int hour, int minute)
static LocalTime of(int hour, int minute, int second)
static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
static LocalTime ofSecondOfDay(long secondOfDay)
```

```
// LocalDate
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
static LocalDate ofYearDay(int year, int dayOfYear)
```

```
// LocalDateTime
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, int month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
                        int minute, int second, int nanoOfSecond)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute)
static LocalDateTime of(int year, Month month, int dayOfMonth,
                        int hour, int minute, int second, int nanoOfSecond)
```

```
static LocalDateTime of(LocalDate date, LocalTime time)
```

```
LocalTime time1 = LocalTime.of(8, 15, 35, 900);   // 08:15:35.000000900
LocalTime time2 = LocalTime.of(16, 45);           // 16:45
// LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
```

```
LocalDate date1 = LocalDate.of(1969, 7, 20);              // 1969-07-20
LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);       // DateTimeException
//  LocalDate date4 = LocalDate.of(2021, 2, 29);        // DateTimeException
// 2021-04-28T12:15
LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
```

```
// 2021-08-19T14:00
LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 19, 14, 0);
```

```
// LocalDate date1 is 1969-07-20.
LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON); // 1969-07-20T12:00
LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);       // 1969-07-20T12:00
LocalDateTime dt5 = date1.atTime(LocalTime.NOON);            // 1969-07-20T12:00
LocalDateTime dt6 = date1.atTime(12, 0);                     // 1969-07-20T12:00
```

```
LocalTime currentTime = LocalTime.now();
LocalDate currentDate = LocalDate.now();
LocalDateTime currentDateTime = LocalDateTime.now();
```

```
// LocalTime
LocalDateTime atDate(LocalDate date)
```

```
// LocalDate
LocalDateTime atTime(LocalTime time)
LocalDateTime atTime(int hour, int minute)
LocalDateTime atTime(int hour, int minute, int second)
LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond)
LocalDateTime atStartOfDay()
```

```
// LocalDateTime
ZonedDateTime atZone(ZoneId zone)
```

```
// LocalTime, LocalDate, LocalDateTime, respectively.
static LocalTime now()
static LocalDate now()
static LocalDateTime now()
```

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;

public class CreatingTemporals {

  public static void main(String[] args) {

    // Creating a specific time from time-based values:
    LocalTime time1 = LocalTime.of(8, 15, 35, 900);// 08:15:35.000000900
    LocalTime time2 = LocalTime.of(16, 45);        // 16:45
//  LocalTime time3 = LocalTime.of(25, 13, 30);    // DateTimeException
    System.out.println("Surveillance start time: " + time1);
    System.out.println("Closing time: " + time2);

    // Creating a specific date from date-based values:
    LocalDate date1 = LocalDate.of(1969, 7, 20);            // 1969-07-20
    LocalDate date2 = LocalDate.of(-3113, Month.AUGUST, 11);// -3113-08-11
//  LocalDate date3 = LocalDate.of(2021, 13, 11);           // DateTimeException
//  LocalDate date4 = LocalDate.of(2021, 2, 29);            // DateTimeException
    System.out.println("Date of lunar landing:        " + date1);
    System.out.println("Start Date of Mayan Calendar: " + date2);

    // Creating a specific date-time from date- and time-based values.
    // 2021-04-28T12:15
    LocalDateTime dt1 = LocalDateTime.of(2021, 4, 28, 12, 15);
    // 2021-08-17T14:00
    LocalDateTime dt2 = LocalDateTime.of(2021, Month.AUGUST, 17, 14, 0);
    System.out.println("Car service appointment: " + dt1);
    System.out.println("Hospital appointment:    " + dt2);

    // Combining date and time objects.
    // 1969-07-20T12:00
    LocalDateTime dt3 = LocalDateTime.of(date1, LocalTime.NOON);
    LocalDateTime dt4 = LocalTime.of(12, 0).atDate(date1);
    LocalDateTime dt5 = date1.atTime(LocalTime.NOON);
    LocalDateTime dt6 = date1.atTime(12, 0);
    System.out.println("Factory date-time combo: " + dt3);
    System.out.println("Time with date combo:    " + dt4);
    System.out.println("Date with time combo:    " + dt5);
    System.out.println("Date with explicit time combo: " + dt6);
    // Current time:
    LocalTime currentTime = LocalTime.now();
    System.out.println("Current time:      " + currentTime);

    // Current date:
    LocalDate currentDate = LocalDate.now();
    System.out.println("Current date:      " + currentDate);

    // Current date and time:
    LocalDateTime currentDateTime = LocalDateTime.now();
    System.out.println("Current date-time: " + currentDateTime);
  }
}
```

```
Surveillance start time: 08:15:35.000000900
Closing time: 16:45
Date of lunar landing:        1969-07-20
Start Date of Mayan Calendar: -3113-08-11
Car service appointment: 2021-04-28T12:15
Hospital appointment:    2021-08-17T14:00
Factory date-time combo: 1969-07-20T12:00
Time with date combo:    1969-07-20T12:00
Date with time combo:    1969-07-20T12:00
Date with explicit time combo: 1969-07-20T12:00
Current time:      10:55:41.296744
Current date:      2021-03-05
Current date-time: 2021-03-05T10:55:41.299318
```

```
// LocalTime, LocalDateTime
int getHour()
int getMinute()
int getSecond()
int getNano()
```

```
// LocalTime, LocalDate, LocalDateTime
int get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

```
int minuteOfHour1 = time.getMinute();                      // 15
int minuteOfHour2 = time.get(ChronoField.MINUTE_OF_HOUR); // 15

int monthVal1 = date.getMonthValue();                      // 8
int monthVal2 = date.get(ChronoField.MONTH_OF_YEAR);       // 8
```

```
LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
LocalDate date = doomsday.toLocalDate();                    // 1945-08-06
LocalTime time = doomsday.toLocalTime();                    // 08:15
```

```
LocalDate foolsday = LocalDate.of(2022, 4, 1);
int daysInMonth = foolsday.lengthOfMonth();     // 30
int daysInYear = foolsday.lengthOfYear();       // 365 (2022 is not a leap year.)
```

```
LocalDate d1 = LocalDate.of(1948, 2, 28);                    // 1948-02-28
LocalDate d2 = LocalDate.of(1949, 3, 1);                     // 1949-03-01
boolean result1 = d1.isBefore(d2);                           // true
boolean result2 = d2.isAfter(d1);                            // true
boolean result3 = d1.isAfter(d1);                            // false
boolean result4 = d1.isEqual(d2);                            // false
boolean result5 = d1.isEqual(d1);                            // true
boolean result6 = d1.isLeapYear();                           // true
```

```
// LocalTime
boolean isBefore(LocalTime other)
boolean isAfter(LocalTime other)
```

```
// LocalDate
boolean isBefore(ChronoLocalDate other)
boolean isAfter(ChronoLocalDate other)
boolean isEqual(ChronoLocalDate other)
boolean isLeapYear()
```

```
// LocalDateTime
boolean isBefore(ChronoLocalDateTime<?> other)
boolean isAfter(ChronoLocalDateTime<?> other)
boolean isEqual(ChronoLocalDateTime<?> other)
```

```
int compareTo(LocalTime other)                  // LocalTime
int compareTo(ChronoLocalDate other)            // LocalDate
int compareTo(ChronoLocalDateTime<?> other)     // LocalDateTime
```

```
// LocalTime, LocalDateTime
LocalTime/LocalDateTime withHour(int hour)
LocalTime/LocalDateTime withMinute(int minute)
LocalTime/LocalDateTime withSecond(int second)
LocalTime/LocalDateTime withNano(int nanoOfSecond)
```

```
// LocalDate, LocalDateTime
LocalDate/LocalDateTime withYear(int year)
LocalDate/LocalDateTime withMonth(int month)
LocalDate/LocalDateTime withDayOfMonth(int dayOfMonth)
LocalDate/LocalDateTime withDayOfYear(int dayOfYear)
```

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime with(TemporalField field, long newValue)
```

```
LocalDate date2 = LocalDate.of(2021, 3, 1);                  // 2021-03-01
date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28); // 2024-02-28
```

```
LocalDate date3 = LocalDate.of(2021, 3, 1);                          // 2021-03-01
date3 = date3
    .with(ChronoField.YEAR, 2024L)
    .with(ChronoField.MONTH_OF_YEAR, 2L)
    .with(ChronoField.DAY_OF_MONTH, 28L);                            // 2024-02-28
```

```
date2 = date2.withYear(2022);                     // 2022-03-01
date2.withMonth(2).withDayOfMonth(28);            // date2 is still 2022-03-01.
```

```
LocalTime time = LocalTime.of(14, 45);       // 14:45
time = time.withMinute(100);       // Out of range. DateTimeException.
time = time.withHour(25);          // Out of range. DateTimeException.

LocalDate date = LocalDate.of(2021, 4, 30);  // 2021-04-30
date = date.withMonth(13);         // Out of range. DateTimeException.
date = date.withDayOfMonth(31);    // Out of range for month. DateTimeException.
date = date.withDayOfYear(366);    // Out of range for year. DateTimeException.
```

```
LocalDate date3 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date3 = date3.withYear(2021);                 // Expected: 2021-02-29
System.out.println("Date3: " + date3);        // Adjusted: 2021-02-28

LocalDate date4 = LocalDate.of(2021, 3, 31);  // Original: 2021-03-31
date4 = date4.withMonth(4);                   // Expected: 2021-04-31
System.out.println("Date4: " + date4);        // Adjusted: 2021-04-30
```

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.temporal.ChronoField;

public class UsingTemporals {

  public static void main(String[] args) {
    // Date-Time: 1945-08-06T08:15
    LocalDateTime doomsday = LocalDateTime.of(1945, 8, 6, 8, 15);
    LocalDate date = doomsday.toLocalDate();                  // 1945-08-06
    LocalTime time = doomsday.toLocalTime();                  // 08:15
    System.out.println("Date-Time: " + doomsday);
    System.out.println();

    // Time: 08:15
    int hourOfDay      = time.getHour();                       // 8
    int minuteOfHour1  = time.getMinute();                     // 15
    int minuteOfHour2  = time.get(ChronoField.MINUTE_OF_HOUR); // 15
    int secondOfMinute = time.getSecond();                     // 0
    System.out.println("Time of day:      " + time);
    System.out.println("Hour-of-day:      " + hourOfDay);
    System.out.println("Minute-of-hour 1: " + minuteOfHour1);
    System.out.println("Minute-of-hour 2: " + minuteOfHour2);
    System.out.println("Second-of-minute: " + secondOfMinute);
    System.out.println();

    // Date: 1945-08-06
    int year       = date.getYear();                           // 1945
    int monthVal1  = date.getMonthValue();                     // 8
    int monthVal2  = date.get(ChronoField.MONTH_OF_YEAR);      // 8
    Month month    = date.getMonth();                          // AUGUST
    DayOfWeek dow  = date.getDayOfWeek();                      // MONDAY
    int day        = date.getDayOfMonth();                     // 6
    System.out.println("Date:  " + date);
    System.out.println("Year:  " + year);
    System.out.println("Month value 1: " + monthVal1);
    System.out.println("Month value 2: " + monthVal2);
    System.out.println("Month-of-year: " + month);
    System.out.println("Day-of-week:   " + dow);
    System.out.println("Day-of-month:  " + day);
    System.out.println();

    // Ordering
    LocalDate d1 = LocalDate.of(1948, 2, 28);                  // 1948-02-28
    LocalDate d2 = LocalDate.of(1949, 3, 1);                   // 1949-03-01
    boolean result1 = d1.isBefore(d2);                         // true
    boolean result2 = d2.isAfter(d1);                          // true
    boolean result3 = d1.isAfter(d1);                          // false
    boolean result4 = d1.isEqual(d2);                          // false
    boolean result5 = d1.isEqual(d1);                          // true
    boolean result6 = d1.isLeapYear();                         // true

    System.out.println("Ordering:");
    System.out.println(d1 + " is before "   + d2 + ": " + result1);
    System.out.println(d2 + " is after "    + d1 + ": " + result2);
    System.out.println(d1 + " is after "    + d1 + ": " + result3);
    System.out.println(d1 + " is equal to " + d2 + ": " + result4);
    System.out.println(d1 + " is equal to " + d1 + ": " + result5);
    System.out.println(d1.getYear() + " is a leap year: " + result6);
    System.out.println();

    System.out.println("Using absolute adjusters:");
    LocalDate date2 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date2);     // 2021-03-01
    date2 = date2.withYear(2024).withMonth(2).withDayOfMonth(28);
    System.out.println("Date after adjusting:  " + date2);     // 2024-02-28
    System.out.println();

    System.out.println("Using temporal fields:");
    LocalDate date3 = LocalDate.of(2021, 3, 1);
    System.out.println("Date before adjusting: " + date3);     // 2021-03-01
    date3 = date3
        .with(ChronoField.YEAR, 2024L)
        .with(ChronoField.MONTH_OF_YEAR, 2L)
        .with(ChronoField.DAY_OF_MONTH, 28L);
    System.out.println("Date after adjusting:  " + date3);     // 2024-02-28
  }
}
```

```
Date-Time: 1945-08-06T08:15

Time of day:      08:15
Hour-of-day:      8
Minute-of-hour 1: 15
Minute-of-hour 2: 15
Second-of-minute: 0

Date:  1945-08-06
Year:  1945
Month value 1: 8
Month value 2: 8
Month-of-year: AUGUST
Day-of-week:   MONDAY
Day-of-month:  6

Ordering:
-1004-03-01 is before 1004-03-01: true
1004-03-01 is after -1004-03-01: true
-1004-03-01 is after -1004-03-01: false
-1004-03-01 is equal to 1004-03-01: false
-1004-03-01 is equal to -1004-03-01: true
1004 is a leap year: true

Using absolute adjusters:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28

Using temporal fields:
Date before adjusting: 2021-03-01
Date after adjusting:  2024-02-28
```

```
// LocalTime, LocalDateTime
LocalTime/LocalDateTime minusHours(long hours)
LocalTime/LocalDateTime plusHours(long hours)

LocalTime/LocalDateTime minusMinutes(long minutes)
LocalTime/LocalDateTime plusMinutes(long minutes)

LocalTime/LocalDateTime minusSeconds(long seconds)
LocalTime/LocalDateTime plusSeconds(long seconds)

LocalTime/LocalDateTime minusNanos(long nanos)
LocalTime/LocalDateTime plusNanos(long nanos)
```

```
// LocalDate, LocalDateTime
LocalDate/LocalDateTime minusYears(long years)
LocalDate/LocalDateTime plusYears(long years)

LocalDate/LocalDateTime minusMonths(long months)
LocalDate/LocalDateTime plusMonths(long months)

LocalDate/LocalDateTime minusWeeks(long weeks)
LocalDate/LocalDateTime plusWeeks(long weeks)

LocalDate/LocalDateTime minusDays(long days)
LocalDate/LocalDateTime plusDays(long days)
```

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(long amountToSub,
                                        TemporalUnit unit)
LocalTime/LocalDate/LocalDateTime plus(long amountToAdd, TemporalUnit unit)
boolean isSupported(TemporalUnit unit)
```

```
// LocalTime, LocalDate, LocalDateTime
LocalTime/LocalDate/LocalDateTime minus(TemporalAmount amountToSub)
LocalTime/LocalDate/LocalDateTime plus(TemporalAmount amountToAdd)
```

```
// LocalTime, LocalDate, LocalDateTime
long until(Temporal endExclusive, TemporalUnit unit)
```

```
// LocalDate
Period until(ChronoLocalDate endDateExclusive)
```

```
LocalDate date = LocalDate.of(2021, 10, 23);               // 2021-10-23
date = date.plusMonths(10).plusWeeks(3).plusDays(40);    // Method chaining
System.out.println(date);                                  // 2022-10-23
date = date.minus(2, ChronoUnit.DAYS)
           .minus(4, ChronoUnit.WEEKS)
           .minus(11, ChronoUnit.MONTHS);                  // Method chaining
System.out.println(date);                                  // 2021-10-23
```

```
LocalTime witchingHour = LocalTime.MIDNIGHT              // 00:00
    .plusHours(14)                                       // 14:00
    .plusMinutes(45)                                     // 14:45
    .plusMinutes(30)                                     // 15:15
    .minusHours(15)                                      // 00:15
    .minusMinutes(15);                                   // 00:00
```

```
LocalDate date5 = LocalDate.of(2020, 2, 29);  // Original: 2020-02-29
date5 = date5.plusYears(1);                   // Expected: 2021-02-29
System.out.println("Date5: " + date5);        // Adjusted: 2021-02-28
```

```
LocalTime busDep = LocalTime.of(12, 15);                  // 12:15
Duration d1 = Duration.ofMinutes(30);                     // PT30M
LocalTime nextBusDep = busDep.plus(d1);                   // 12:45

LocalDate birthday = LocalDate.of(2020, 10, 23);          // 2020-10-23
Period p1 = Period.ofYears(1);                            // P1Y
LocalDate nextBirthday = birthday.plus(p1);               // 2021-10-23
LocalDate currentDate = LocalDate.now();
LocalDate newYearDay = currentDate.plusYears(1).withMonth(1).withDayOfMonth(1);
long daysToNewYear = currentDate.until(newYearDay, ChronoUnit.DAYS); // (1)
System.out.println("Current Date: " + currentDate); // Current Date: 2021-03-08
System.out.println("New Year's Day: " + newYearDay);// New Year's Day: 2022-01-01
System.out.println("Days to New Year: " + daysToNewYear);// Days to New Year: 299
```

```
long minsToMidnight = LocalDateTime.now()             // (1) DateTimeException!
         .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

```
long minsToMidnight = LocalTime.now()                  // (2)
        .until(LocalTime.MIDNIGHT.minusSeconds(1), ChronoUnit.MINUTES);
```

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class TemporalArithmetic {

  public static void main(String[] args) {

    LocalDate date = LocalDate.of(2021, 10, 23);              // (1)
    System.out.println("Date:                " + date);       // 2021-10-23
    date = date.plusMonths(10);                               // (2)
    System.out.println("10 months after:  " + date);          // 2022-08-23
    date = date.plusWeeks(3);                                 // (3)
    System.out.println("3 weeks after:    " + date);          // 2022-09-13
    date = date.plusDays(40);                                   // (4)
    System.out.println("40 days after:    " + date);            // 2022-10-23

    date = date.minus(2, ChronoUnit.DAYS);                      // (5)
    System.out.println("2 days before:    " + date);            // 2022-10-21
    date = date.minus(4, ChronoUnit.WEEKS);                     // (6)
    System.out.println("4 weeks before:   " + date);            // 2022-09-23
    date = date.minus(11, ChronoUnit.MONTHS);                   // (7)
    System.out.println("11 months before: " + date);            // 2021-10-23
  }
}
```

```
boolean isSupportedBy(Temporal temporal)
```

`static ChronoUnit[] values()`

```
LocalDate date = LocalDate.of(2021, 10, 23);
System.out.print("Date " + date);
date = date.minus(10, ChronoUnit.MONTHS).minus(3, ChronoUnit.DAYS);
System.out.println(" minus 10 months and 3 days: " + date);
// Date 2021-10-23 minus 10 months and 3 days: 2020-12-20

LocalTime time = LocalTime.of(14, 15);
System.out.print("Time " + time);
time = time.plus(9, ChronoUnit.HOURS).plus(70, ChronoUnit.MINUTES);
System.out.println(" plus 9 hours and 70 minutes is " + time);
// Time 14:15 plus 9 hours and 70 minutes is 00:25
```

```
LocalDate fromDate = LocalDate.of(2021, 3, 1);
LocalDate xmasDate = LocalDate.of(2021, 12, 25);
long tilChristmas = fromDate.until(xmasDate, ChronoUnit.DAYS);
System.out.println("From " + fromDate + ", days until Xmas: " + tilChristmas);
// From 2021-03-01, days until Xmas: 299
```

```
boolean isSupportedBy(TemporalAccessor temporal)
```

```
LocalDate date = LocalDate.of(2021, 8, 13);
int monthValue = date.get(ChronoField.MONTH_OF_YEAR);
System.out.print("Date " + date + " has month of the year: " + monthValue);
// Date 2021-08-13 has month of the year: 8
```

```
LocalDateTime dateTime = LocalDateTime.of(2021, 8, 13, 20, 20);
System.out.print("Date-time " + dateTime);
dateTime = dateTime.with(ChronoField.DAY_OF_MONTH, 11)
                   .with(ChronoField.MONTH_OF_YEAR, 1)
                   .with(ChronoField.YEAR, 2022);
```

```
System.out.println(" changed to: " + dateTime);
// Date-time 2021-08-13T20:20 changed to: 2022-01-11T20:20
```

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class ValidTemporalUnitsAndFields {

  public static void main(String[] args) {

    // Temporals:
    LocalTime time = LocalTime.now();
    LocalDate date = LocalDate.now();
    LocalDateTime dateTime = LocalDateTime.now();
    ZonedDateTime zonedDateTime = ZonedDateTime.now();
    Instant instant = Instant.now();

    // Print supported units:                                             // (1)
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoUnit", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoUnit[] units =  ChronoUnit.values();                            // (2)
    for (ChronoUnit unit : units) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          unit.name(), time.isSupported(unit), date.isSupported(unit),    // (3)
          dateTime.isSupported(unit), zonedDateTime.isSupported(unit),    // (4)
          instant.isSupported(unit));                                     // (5)
    }
    System.out.println();

    // Print supported fields:                                            // (6)
    System.out.printf("%29s %s %s %s %s %s%n",
        "ChronoField", "LocalTime", "LocalDate", "LocalDateTime",
        " ZDT ", "Instant");
    ChronoField[] fields =  ChronoField.values();                         // (7)
    for (ChronoField field : fields) {
      System.out.printf("%28S: %7b %9b %10b %9b %7b%n",
          field.name(), time.isSupported(field), date.isSupported(field),// (8)
          dateTime.isSupported(field), zonedDateTime.isSupported(field), // (9)
          instant.isSupported(field));                                   // (10)
    }
    System.out.println();
  }
}
```

```
                   ChronoUnit LocalTime LocalDate LocalDateTime  ZDT  Instant
                       NANOS:    true     false       true      true    true
                      MICROS:    true     false       true      true    true
                      MILLIS:    true     false       true      true    true
                     SECONDS:    true     false       true      true    true
                     MINUTES:    true     false       true      true    true
                       HOURS:    true     false       true      true    true
                   HALF_DAYS:    true     false       true      true    true
                        DAYS:   false      true       true      true    true
                       WEEKS:   false      true       true      true   false
                      MONTHS:   false      true       true      true   false
                       YEARS:   false      true       true      true   false
                     DECADES:   false      true       true      true   false
                   CENTURIES:   false      true       true      true   false
                   MILLENNIA:   false      true       true      true   false
                        ERAS:   false      true       true      true   false
                     FOREVER:   false     false      false     false   false

                  ChronoField LocalTime LocalDate LocalDateTime  ZDT  Instant
              NANO_OF_SECOND:    true     false       true      true    true
                 NANO_OF_DAY:    true     false       true      true   false
             MICRO_OF_SECOND:    true     false       true      true    true
                MICRO_OF_DAY:    true     false       true      true   false
             MILLI_OF_SECOND:    true     false       true      true    true
                MILLI_OF_DAY:    true     false       true      true   false
            SECOND_OF_MINUTE:    true     false       true      true   false
               SECOND_OF_DAY:    true     false       true      true   false
              MINUTE_OF_HOUR:    true     false       true      true   false
               MINUTE_OF_DAY:    true     false       true      true   false
                HOUR_OF_AMPM:    true     false       true      true   false
          CLOCK_HOUR_OF_AMPM:    true     false       true      true   false
                 HOUR_OF_DAY:    true     false       true      true   false
           CLOCK_HOUR_OF_DAY:    true     false       true      true   false
                 AMPM_OF_DAY:    true     false       true      true   false
                 DAY_OF_WEEK:   false      true       true      true   false
ALIGNED_DAY_OF_WEEK_IN_MONTH:   false      true       true      true   false
 ALIGNED_DAY_OF_WEEK_IN_YEAR:   false      true       true      true   false
                DAY_OF_MONTH:   false      true       true      true   false
                 DAY_OF_YEAR:   false      true       true      true   false
                   EPOCH_DAY:   false      true       true      true   false
       ALIGNED_WEEK_OF_MONTH:   false      true       true      true   false
        ALIGNED_WEEK_OF_YEAR:   false      true       true      true   false
               MONTH_OF_YEAR:   false      true       true      true   false
             PROLEPTIC_MONTH:   false      true       true      true   false
                 YEAR_OF_ERA:   false      true       true      true   false
                        YEAR:   false      true       true      true   false
            ERA:   false      true      true      true   false
INSTANT_SECONDS:   false     false     false      true    true
 OFFSET_SECONDS:   false     false     false      true   false
```

```
static Instant ofEpochMilli(long epochMilli)
static Instant ofEpochSecond(long epochSecond)
static Instant ofEpochSecond(long epochSecond, long nanoAdjustment)
```

```
uuuu-MM-ddTHH:mm:ss.SSSSSSSSSZ
```

```
static Instant parse(CharSequence text)
```

```
ZonedDateTime atZone(ZoneId zone)
```

```
Instant currentInstant = Instant.now();      // 2021-03-09T10:48:01.914826Z
Instant inst1 = Instant.ofEpochMilli(-24L*60*60*1000);// Date 1 day before epoch.
Instant inst2 = Instant.ofEpochSecond(24L*60*60);     // Date 1 day after epoch.
Instant inst3 = Instant.ofEpochSecond(24L*60*60 - 1,  // Date 1 day after epoch.
                                      1_000_000_000L);
out.println("A day before: " + inst1); // Date 1 day before: 1969-12-31T00:00:00Z
out.println("A day after:  " + inst2); // Date 1 day after : 1970-01-02T00:00:00Z
out.println("A day after:  " + inst3); // Date 1 day after : 1970-01-02T00:00:00Z
```

```
Instant inst4 = Instant.ofEpochSecond(0, 500);
out.println("Default format:  " + inst4);       // 1970-01-01T00:00:00.000000500Z
```

```
Instant instA = Instant.parse("1970-01-01T00:00:00.000000500Z");
Instant instB = Instant.parse("1949-03-01T12:30:15Z");
Instant instC = Instant.parse("-1949-03-01T12:30:15Z");
Instant instD = Instant.parse("-1949-03-01T12:30:15"); // DateTimeParseException!
```

```
LocalDateTime ldt = LocalDate.of(2021, 12, 25).atStartOfDay();  //(1)
Instant i1 = ldt.toInstant(ZoneOffset.of("+02:00"));     // (2) Ahead of UTC
Instant i2 = ldt.toInstant(ZoneOffset.UTC);              // (3) At UTC
Instant i3 = ldt.toInstant(ZoneOffset.of("-02:00"));     // (4) Behind UTC
System.out.println("ldt: " + ldt);
System.out.println("i1:  " + i1);
System.out.println("i2:  " + i2);
System.out.println("i3:  " + i3);
```

```
// LocalDateTime
default Instant toInstant(ZoneOffset offset)
```

```
int  get(TemporalField field)
long getLong(TemporalField field)
```

```
boolean isSupported(TemporalField field)
```

```
Instant inst = Instant.ofEpochSecond(24L*60*60,    // 1 day and
                                     555_555_555L);// 555555555 ns after epoch.
out.println(inst);                   // 1970-01-02T00:00:00.555555555Z
out.println(inst.getNano());         // 555555555 ns
out.println(inst.getEpochSecond());  // 86400 s
out.println(inst.get(ChronoField.NANO_OF_SECOND));      // 555555555 ns
out.println(inst.get(ChronoField.MICRO_OF_SECOND));     // 555555 micros
out.println(inst.get(ChronoField.MILLI_OF_SECOND));      // 555 ms
out.println(inst.getLong(ChronoField.INSTANT_SECONDS)); // 86400 s
//out.println(inst.get(ChronoField.INSTANT_SECONDS));   // DateTimeException
//out.println(inst.get(ChronoField.HOUR_OF_DAY));       // UnsupportedTemporal-
                                                        // TypeException
```

```
out.println(inst.toEpochMilli());                          // 86400555 ms
```

```
// instA is  1970-01-01T00:00:00.000000500Z
// instB is  1949-03-01T12:30:15Z
// instC is -1949-03-01T12:30:15Z
out.println(instA.isBefore(instB));              // false
out.println(instA.isAfter(instC));               // true
out.println(instA.equals(instB));                  // false
out.println(instA.equals(instC));                  // false

List<Instant> list = Arrays.asList(instA, instB, instC);
Collections.sort(list);             // Natural order: position on the timeline.
// [-1949-03-01T12:30:15Z, 1949-03-01T12:30:15Z, 1970-01-01T00:00:00.000000500Z]
```

```
boolean isBefore(Instant other)
boolean isAfter(Instant other)
```

```
Instant with(TemporalField field, long newValue)
Instant i0, i1, i2, i3;
i0 = Instant.now();
out.println(i0);                                    // 2021-02-28T08:43:35.864Z
i1 = i0.with(ChronoField.NANO_OF_SECOND,  500_000_000);// 500000000 ns.
i2 = i0.with(ChronoField.MICRO_OF_SECOND, 500_000);    // 500000x1000 ns.
i3 = i0.with(ChronoField.MILLI_OF_SECOND, 500);        // 500x1000000 ns.
out.println(i1);                                    // 2021-02-28T08:43:35.500Z

out.println(i1.equals(i2));                         // true
out.println(i1.equals(i3));                         // true
```

```
Instant oneInstant = Instant.now()
                            .with(ChronoField.MILLI_OF_SECOND, 500)
                            .with(ChronoField.INSTANT_SECONDS, 24L*60*60);
out.println(oneInstant);                     // 1970-01-02T00:00:00.500Z
```

```
Instant event =
    Instant.EPOCH                    //             1970-01-01T00:00:00Z
           .plusSeconds(7L*24*60*60) // (+7days)    1970-01-08T00:00:00Z
           .plusSeconds(6L*60*60)    // (+6hrs)     1970-01-08T06:00:00Z
           .plusSeconds(5L*60)       // (+5mins)    1970-01-08T06:05:00Z
           .plusSeconds(4L)          // (+4s)       1970-01-08T06:05:04Z
           .plusMillis(3L*100)       // (+3ms)      1970-01-08T06:05:04.003Z
           .plusNanos(2L*1_000)      // (+2micros) 1970-01-08T06:05:04.003002Z
           .plusNanos(1L);           // (+1ns)      1970-01-08T06:05:04.003002001Z
```

```
Instant ptInTime =
     Instant.EPOCH                            // 1970-01-01T00:00:00Z
            .plus(7L, ChronoUnit.DAYS)        // 1970-01-08T00:00:00Z
            .plus(6L, ChronoUnit.HOURS)       // 1970-01-08T06:00:00Z
            .plus(5L, ChronoUnit.MINUTES)     // 1970-01-08T06:05:00Z
            .plus(4L, ChronoUnit.SECONDS)     // 1970-01-08T06:05:04Z
            .plus(3L, ChronoUnit.MILLIS)      // 1970-01-08T06:05:04.003Z
            .plus(2L, ChronoUnit.MICROS)      // 1970-01-08T06:05:04.003002Z
            .plus(1L, ChronoUnit.NANOS);      // 1970-01-08T06:05:04.003002001Z
```

```
Instant start = Instant.EPOCH
                       .plus(20, ChronoUnit.MINUTES);// 1970-01-01T00:20:00Z
Duration length = Duration.ZERO.plusMinutes(90);     // PT1H30M (90 mins)
Instant end = start.plus(length);                    // 1970-01-01T01:50:00Z
```

```
long eventDuration1 = start.until(end, ChronoUnit.MINUTES);  // 90 minutes
long eventDuration2 = start.until(end, ChronoUnit.HOURS);    // 1 hour
```

```
Instant plusSeconds/minusSeconds(long seconds)
Instant plusMillis/minusMillis(long millis)
Instant plusNanos/minusNanos(long nanos)
Instant plus(long amountToAdd, TemporalUnit unit)
Instant minus(long amountToSub, TemporalUnit unit)
Instant isSupported(TemporalUnit unit)
Instant plus(TemporalAmount amountToAdd)
Instant minus(TemporalAmount amountToSubtract)
```

```
long until(Temporal endExclusive, TemporalUnit unit)
```

```
Instant instant = Instant.parse("2021-04-28T03:15:00Z");
ZoneId zid = ZoneId.of("America/New_York");
LocalTime lt = LocalTime.ofInstant(instant, zid);           // 10:18:30
LocalDate ld = LocalDate.ofInstant(instant, zid);           // 2021-04-27
LocalDateTime ldt = LocalDateTime.ofInstant(instant, zid);  // 2021-04-27T23:15
ZonedDateTime zdt = ZonedDateTime.ofInstant(instant, zid);
    // 2021-04-27T23:15-04:00[America/New_York]
```

```
static TemporalType ofInstant(Instant instant, ZoneId zone)
```

```
Period p = Period.of(2, 4, 8);          // (1)
System.out.println(p);                  // (2) P2Y4M8D (2 Years, 4 Months, 8 Days)
Period p1 = Period.ofYears(10);         // P10Y, period of 10 years.
Period p2 = Period.ofMonths(14);        // P14M, period of 14 months.
Period p3 = Period.ofDays(40);          // P40D, period of 40 days.
Period p4 = Period.ofWeeks(2);          // P14D, period of 14 days (2 weeks).
Period period = Period.ofYears(3).ofMonths(4).ofDays(5);   // P5D. Logical error.
```

```
LocalDate d1 = LocalDate.of(2021, 3, 1);  // 2021-03-01
LocalDate d2 = LocalDate.of(2022, 3, 1);  // 2022-03-01
Period period12 = Period.between(d1, d2); // P1Y
Period period21 = Period.between(d2, d1); // P-1Y
Period period2 = Period.parse("P1Y15M20D"); // 1 year, 15 months, 20 days
Period period3 = Period.parse("P20D");      // 20 days
Period period4 = Period.parse("P5W");       // 35 days (5 weeks)
//  Period pX = Period.parse("P24H"); // java.time.format.DateTimeParseException
```

```
static Period of(int years, int months, int days)
static Period ofYears(int years)
static Period ofMonths(int months)
static Period ofWeeks(int weeks)
static Period ofDays(int days)
static Period between(LocalDate startDateInclusive,
                      LocalDate endDateExclusive)
```

```
static Period parse(CharSequence text)
```

```
Period period5 = Period.of(2, 4, -10);
System.out.println("Period: " + period5);             // Period: P2Y4M-10D
System.out.println("Years:  " + period5.getYears());  // Years:  2
System.out.println("Months: " + period5.getMonths()); // Months: 4
System.out.println("Days:   " + period5.getDays());   // Days:   -10
System.out.println("Years:  " + period5.get(ChronoUnit.YEARS)); // Years:  2
System.out.println("Months: " + period5.get(ChronoUnit.MONTHS));// Months: 4
System.out.println("Days:   " + period5.get(ChronoUnit.DAYS));  // Days: -10
List<TemporalUnit> supportedUnits = period5.getUnits(); // [Years, Months, Days]

System.out.println("Total months: " + period5.toTotalMonths()); // 28 months
System.out.println(period5.isNegative());                       // true
System.out.println(period5.isZero());                           // false
```

```
Period px = Period.of(1, 14, 0);
Period py = Period.of(2, 2, 0);
Period pz = Period.ofMonths(26);
System.out.println(px.equals(py));           // false
System.out.println(px.equals(pz));           // false
System.out.println(px.equals(Period.ZERO));  // false
```

```
Period p5 = Period.of(2, 1, 30) // P2Y1M30D
    .withYears(3)               // P3Y1M30D, sets the number of years
    .withMonths(16)             // P3Y16M30D, sets the number of months
    .withDays(1);               // P3Y16M1D, sets the number of days
```

```
Period p6 = Period.of(2, 10, 30)  // P2Y10M30D
    .plusDays(10)                 // P2Y10M40D
    .plusMonths(8)                // P2Y18M40D
    .plusYears(1)                 // P3Y18M40D
    .normalized();                // P4Y6M40D
```

```
Period p7 = Period.of(1, 1, 1);                    // P1Y1M1D
Period p8 = Period.of(2, 12, 30);                  // P2Y12M30D
Period p9 = p8.minus(p7);                          // P1Y11M29D
p8 = p8.plus(p7).plus(p8);                         // P3Y13M31D, P5Y25M61D
```

```
Period plusYears/minusYears(long years)
Period plusMonths/minusMonths(long months)
Period plusDays/minusDays(long days)
Period plus(TemporalAmount amount)
Period minus(TemporalAmount amount)
Period negated()
Period multipliedBy(int scalar)
Period p10 = Period.of(1, 1, 1);                // P1Y1M1D
LocalDate date1 = LocalDate.of(2021, 3, 1);     // 2021-03-01
LocalDate date2 = date1.plus(p10);              // 2022-04-02
LocalDate date3 = date1.minus(p10);             // 2020-01-31
```

```
LocalTime time = LocalTime.NOON;
time = time.plus(p10);     // java.time.temporal.UnsupportedTemporalTypeException
```

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodBasedLoop {
  public static void main(String[] args) {
    reserveDates(Period.ofDays(7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2021, 11, 20));
    System.out.println();
    reserveDates(Period.ofMonths(1),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 20));
    System.out.println();
    reserveDates(Period.of(0, 1, 7),
                 LocalDate.of(2021, 10, 20), LocalDate.of(2022, 1, 21));
  }

  public static void reserveDates(Period period,                  // (1)
                                  LocalDate fromDate,
                                  LocalDate toDateExclusive) {
    System.out.println("Start date: " + fromDate);
    for (LocalDate date = fromDate.plus(period);                  // (2)
         date.isBefore(toDateExclusive);
         date = date.plus(period)) {
      System.out.println("Reserved (" + period + "): " + date);
    }
    System.out.println("End date: " + toDateExclusive);
  }
}
```

```
Start date: 2021-10-20
Reserved (P7D): 2021-10-27
Reserved (P7D): 2021-11-03
Reserved (P7D): 2021-11-10
Reserved (P7D): 2021-11-17
End date: 2021-11-20

Start date: 2021-10-20
Reserved (P1M): 2021-11-20
Reserved (P1M): 2021-12-20
End date: 2022-01-20

Start date: 2021-10-20
Reserved (P1M7D): 2021-11-27
Reserved (P1M7D): 2022-01-03
End date: 2022-01-21
```

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;

public class ActYourAge {

  public static void main(String[] args) {
    birthdayInfo(LocalDate.of(1981, Month.AUGUST, 19));
    birthdayInfo(LocalDate.of(1935, Month.JANUARY, 8));
  }

  public static void birthdayInfo(LocalDate dateOfBirth) {           // (1)
    LocalDate today = LocalDate.now();
    System.out.println("Today:         " + today);
    System.out.println("Date of Birth: " + dateOfBirth);
    Period p1 = Period.between(dateOfBirth, today);                  // (2)
    System.out.println("Age:           " +
                             p1.getYears()  + " years, " +
                             p1.getMonths() + " months, and " +
                             p1.getDays()   + " days");

    LocalDate nextBirthday =  dateOfBirth.withYear(today.getYear()); // (3)
    if (nextBirthday.isBefore(today) ||                              // (4)
        nextBirthday.isEqual(today)) {
      nextBirthday = nextBirthday.plusYears(1);                      // (5)
    }
    Period p2 = today.until(nextBirthday);                           // (6)
    System.out.println("Birthday in " + p2.getMonths() + " months and " +
                                    p2.getDays()   + " days");
  }
}
```

```
Today:         2021-03-05
Date of Birth: 1981-08-19
Age:           39 years, 6 months, and 14 days
Birthday in 5 months and 14 days
Today:         2021-03-05
Date of Birth: 1935-01-08
Age:           86 years, 1 months, and 25 days
Birthday in 10 months and 3 days
```

```
Duration d1 = Duration.ofDays(1L);                                  // PT24H
Duration d2 = Duration.ofHours(24L);                                // PT24H
Duration d3 = Duration.ofMinutes(24L*60);                           // PT24H
Duration d4 = Duration.ofSeconds(24L*60*60);                        // PT24H
Duration d5 = Duration.ofMillis(24L*60*60*1000);                    // PT24H
Duration d6 = Duration.ofSeconds(24L*60*60 - 1, 1_000_000_000L); // (1) PT24H
Duration d7 = Duration.ofNanos(24L*60*60*1_000_000_000); // (2) PT24H
Duration d8 = Duration.ofNanos(24*60*60*1_000_000_000);  // (3) PT-1.857093632S
Duration d11 = Duration.of(1L, ChronoUnit.DAYS);              // P24H
Duration d22 = Duration.of(24L, ChronoUnit.HOURS);            // P24H
Duration d33 = Duration.of(24L*60, ChronoUnit.MINUTES);       // P24H
Duration d44 = Duration.of(24L*60*60, ChronoUnit.SECONDS);    // P24H
Duration d88 = Duration.of(24L*60*60*1000, ChronoUnit.MILLIS);// P24H
Duration d77 = Duration.of(24L*60*60*1_000_000_000,
                           ChronoUnit.NANOS);                 // P24H
```

```
Duration duration = Duration.ofDays(7).ofHours(24);   // PT24H. Logical error.
```

```
LocalTime startTime = LocalTime.of(14, 30);                 // 14:30
LocalTime endTime   = LocalTime.of(17, 45, 15);             // 17:45:15
```

```
Duration interval1 = Duration.between(startTime, endTime);   // PT3H15M15S
Duration interval2 = Duration.between(endTime, startTime);   // PT-3H-15M-15S
```

```
LocalDateTime dateTime = LocalDateTime.of(2021, 4, 28,
                                          17, 45, 15);    // 2021-04-28T17:45:15
Duration interval3 = Duration.between(startTime, dateTime);  // PT3H15M15S
Duration interval4 = Duration.between(dateTime, startTime);  // DateTimeException!
Duration da = Duration.parse("PT3H15M10.1S");// 3hrs. 15mins. 10.1s.(PT3H15M10.1S)
Duration db = Duration.parse("PT0.999S");    // 999000000 nanos.    (PT0.999S)
Duration dc = Duration.parse("-PT30S");      // -30 seconds.        (PT-30S)
Duration dd = Duration.parse("P-24D");       // -24 days            (PT-576H)
Duration dd = Duration.parse("P24H");        // Missing T. DateTimeParseException!
```

```
static Duration ofDays(long days)
static Duration ofHours(long hours)
static Duration ofMinutes(long minutes)
static Duration ofMillis(long millis)
static Duration ofSeconds(long seconds)
static Duration ofSeconds(long seconds, long nanoAdjustment)
static Duration ofNanos(long nanos)
static Duration of(long amount, TemporalUnit unit)
static Duration between(Temporal startInclusive, Temporal endExclusive)
static Duration parse(CharSequence text)
```

```
Duration dx = Duration.ofSeconds(12L*60*60, 500_000_000L); // PT12H0.5S
out.println(dx.getNano());                                 // 500000000
out.println(dx.getSeconds());                              // 43200 (i.e. 12 hrs.)
out.println(dx.get(ChronoUnit.NANOS));          // 500000000
out.println(dx.get(ChronoUnit.SECONDS));        // 43200
out.println(dx.get(ChronoUnit.MINUTES));        // UnsupportedTemporalTypeException
out.println(dx.getUnits());                     // [Seconds, Nanos]
out.println("Days:    " + dx.toDays());      // Days:    0
out.println("Hours:   " + dx.toHours());     // Hours:   12
out.println("Minutes: " + dx.toMinutes());   // Minutes: 720
out.println("Millis:  " + dx.toMillis());    // Millis:  43200500
out.println("Nanos:   " + dx.toNanos());     // Nanos:   43200500000000
```

```
Duration eatBreakFast = Duration.ofMinutes(20L);                // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);             // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES);   // PT45M

out.println(eatBreakFast.equals(eatLunch));                     // false
out.println(Duration.ofSeconds(0).equals(Duration.ZERO));       // true

List<Duration> ld = Arrays.asList(eatSupper, eatBreakFast, eatLunch );
Collections.sort(ld);                                  // Natural order.
out.println(ld);                                       // [PT20M, PT30M, PT45M]
```

```
boolean equals(Object otherDuration)
```

```
int compareTo(Duration otherDuration)
```

```
Duration oneDuration =  Duration.ZERO                        // PT0S
                                .withNanos(500_000_000)    // New copy: PT0.5S
                                .withSeconds(12L*60*60);   // New copy: PT12H0.5S
```

```
Duration withNanos(int nanoOfSecond)
Duration withSeconds(long seconds)
Duration max20H = Duration.ZERO                         // PT0S
                          .plusHours(10)                // PT10H
                          .plusMinutes(10*60 + 30)      // PT20H30M
                          .plusSeconds(6*60*60 + 15)    // PT26H30M15S
                          .minusMinutes(2*60 + 30)      // PT24H15S
                          .minusSeconds(15);            // PT24H
```

```
Duration max20H2 =
    Duration.ZERO                                         // PT0S
            .plus(10L,           ChronoUnit.HOURS)        // PT10H
            .plus(10*60 + 30,    ChronoUnit.MINUTES)      // PT20H30M
            .plus(6*60*60L + 15, ChronoUnit.SECONDS)      // PT26H3015S
            .minus(2*60 + 30,    ChronoUnit.MINUTES)      // PT24H15S
            .minus(15,           ChronoUnit.SECONDS);     // PT24H
```

```
Duration eatBreakFast = Duration.ofMinutes(20L);                // PT20M
Duration eatLunch     = Duration.ofSeconds(30L*60);             // PT30M
Duration eatSupper    = Duration.of(45L, ChronoUnit.MINUTES);   // PT45M
Duration totalTimeForMeals = eatBreakFast                       // PT20M
    .plus(eatLunch)                                             // PT50M
    .plus(eatSupper);                                           // PT1H35M
```

```
Duration result = Duration.ofSeconds(-100, -500_000_000) // -100.5 => PT-1M-40.5S
                          .abs()           // abs(-100.5) = 100.5  => PT1M40.5S
                          .multipliedBy(4) // 100.5*4 = 402 => PT6M42S
                          .dividedBy(2);   // 402 / 2 = 201 => PT3M21S
```

```
Duration plusDays/minusDays(long days)
Duration plusHours/minusHours(long hours)
Duration plusMinutes/minusMinutes(long minutes)
Duration plusSeconds/minusSeconds(long seconds)
Duration plusMillis/minusMillis(long millis)
Duration plusNanos/minusNanos(long nanos)
Duration plus(long amountToAdd, TemporalUnit unit)
Duration minus(long amountToSub, TemporalUnit unit)
Duration plus(Duration duration)
Duration minus(Duration duration)
Duration dividedBy(long divisor)
Duration multipliedBy(long multiplicand)
```

```
LocalTime timeA = LocalTime.of(14,45,30);                  // 14:45:30
LocalDate dateA = LocalDate.of(2021, 4, 28);               // 2021-04-28
LocalDateTime dateTimeA = LocalDateTime.of(dateA, timeA); // 2021-04-28T14:45:30

Duration amount = Duration.ofMinutes(20);                  // PT20M

timeA = timeA.plus(amount);                                // 15:05:30
dateTimeA = dateTimeA.minus(amount);                       // 2021-04-28T14:25:30

dateA = dateA.minus(amount);                // UnsupportedTemporalTypeException
```

```
import java.time.LocalTime;
import java.time.Duration;

public class DurationBasedLoop {
  public static void main(String[] args) {
    Duration duration = Duration.ofHours(2).plusMinutes(15);     // PT2H15M
    LocalTime firstShowTime = LocalTime.of(10, 10);              // 10:10
    LocalTime endTimeExclusive = LocalTime.of(23, 0);            // 23:00
    for (LocalTime time = firstShowTime;                         // (1)
         time.plus(duration).isBefore(endTimeExclusive);
         time = time.plus(duration)) {
      System.out.println("Showtime (" + duration + "): " + time);
    }
    System.out.println("Closing time: " + endTimeExclusive);
  }
}
```

```
ZoneId.getAvailableZoneIds()                 // Prints a long list of zone names.
      .stream()
      .sorted()
      .forEachOrdered(System.out::println);  // Output not shown intentionally.
```

```
ZoneId estZID = ZoneId.of("US/Eastern");              // Create a time zone ID.
System.out.println(estZID);                           // US/Eastern
System.out.println(ZoneId.systemDefault());           // Europe/Oslo
```

```
static ZoneId of(String zoneId)
```

```
String          toString()
abstract String getId()
```

```
static Set<String> getAvailableZoneIds()
```

```
ZonedDateTime defaultZDT = ZonedDateTime.now();                          // (1a)
ZonedDateTime utcZDT     = ZonedDateTime.now(ZoneId.of("UTC"));          // (2a)
ZonedDateTime edtZDT     = ZonedDateTime.now(ZoneId.of("US/Eastern"));   // (3a)
System.out.println("Default Zone Date-time:  " + defaultZDT);
System.out.println("UTC Date-time:           " + utcZDT);
System.out.println("EDT Zone Date-time:      " + edtZDT);
```

```
Default Zone Date-time: 2021-07-11T11:35:20.008+02:00[Europe/Oslo]       // (1b)
UTC Date-time:          2021-07-11T09:35:20.023Z[UTC]                     // (2b)
EDT Zone Date-time:     2021-07-11T05:35:20.023-04:00[US/Eastern]         // (3b)
```

```
LocalTime concertTime = LocalTime.of(00, 10);                    // 00:10
LocalDate concertDate = LocalDate.of(1973, Month.JANUARY, 14); // 1973-01-14
LocalDateTime concertDT = LocalDateTime.of(concertDate,
                                          concertTime);  // 1973-01-14T00:10
ZoneId hwZID = ZoneId.of("US/Hawaii");                   // US/Hawaii
Instant instantZ = Instant.ofEpochSecond(95854200);       // 1973-01-14T10:10:00Z
```

```
ZonedDateTime concertZDT0 = ZonedDateTime.of(concertDate, concertTime, hwZID);
ZonedDateTime concertZDT1 = ZonedDateTime.of(concertDT, hwZID);
ZonedDateTime concertZDT2 = ZonedDateTime.ofInstant(instantZ, hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
boolean areEqual = concertZDT0.equals(concertZDT1)
                && concertZDT0.equals(concertZDT2);      // true
```

```
ZonedDateTime concertZDT3 = concertDT.atZone(hwZID);
ZonedDateTime concertZDT4 = instantZ.atZone(hwZID);
// 1973-01-14T00:10-10:00[US/Hawaii]
```

```
ZonedDateTime concertZDT5
    = ZonedDateTime.parse("1973-01-14T00:10-10:00[US/Hawaii]");
```

```
static ZonedDateTime now()
static ZonedDateTime now(ZoneId zone)
static ZonedDateTime of(int year, int month, int dayOfMonth, int hour,
                        int minute, int second, int nanoOfSecond, ZoneId zone)
static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)
static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
static ZonedDateTime ofInstant(Instant instant, ZoneId zone)
static ZonedDateTime parse(CharSequence text)
```

```
// Using ChronoField constants:
int theDay = concertZDT0.get(ChronoField.DAY_OF_MONTH);          // 14
int theMonthValue = concertZDT0.get(ChronoField.MONTH_OF_YEAR);  // 1
int theYear = concertZDT0.get(ChronoField.YEAR);                 // 1973

// Using specific get methods:
int theMonthValue2 = concertZDT0.getMonthValue();                // 1
Month theMonth     = concertZDT0.getMonth();                     // JANUARY

// Extracting constituent parts:
LocalTime theTime     = concertZDT0.toLocalTime();         // 00:10
LocalDate theDate     = concertZDT0.toLocalDate();         // 1973-01-14
LocalDateTime theDT   = concertZDT0.toLocalDateTime();     // 1973-01-14T00:10
ZoneId theZID         = concertZDT0.getZone();             // US/Hawaii
ZoneOffset theZoffset = concertZDT0.getOffset();           // -10:00
```

`DayOfWeek`	`getDayOfWeek()`
`Month`	`getMonth()`

```
int  get(TemporalField field)
long getLong(TemporalField field)
boolean isSupported(TemporalField field)
```

`LocalTime`	`toLocalTime()`
`LocalDate`	`toLocalDate()`
`LocalDateTime`	`toLocalDateTime()`

```
ZonedDateTime theZDT = concertZDT0         // 1973-01-14T00:10-10:00[US/Hawaii]
    .withYear(1977)                        // 1977-01-14T00:10-10:00[US/Hawaii]
    .with(ChronoField.MONTH_OF_YEAR, 8)    // 1977-08-14T00:10-10:00[US/Hawaii]
    .withDayOfMonth(16)                    // 1977-08-16T00:10-10:00[US/Hawaii]
    .with(ChronoField.HOUR_OF_DAY, 9)      // 1977-08-16T09:10-10:00[US/Hawaii]
    .with(ChronoField.MINUTE_OF_HOUR, 30); // 1977-08-16T09:30-10:00[US/Hawaii]
```

```
ZoneId cTZ = ZoneId.of("US/Central");
ZonedDateTime zdtSameLocal = theZDT.withZoneSameLocal(cTZ);             // (1)
```

```
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameLocal, zdtSameLocal.toInstant());
```

```
          ZonedDateTime                     Instant
1977-08-16T09:30-10:00[US/Hawaii]    1977-08-16T19:30:00Z
1977-08-16T09:30-05:00[US/Central]   1977-08-16T14:30:00Z
```

```
ZonedDateTime zdtSameInstant = theZDT.withZoneSameInstant(cTZ);        // (2)
System.out.printf("%23s %25s%n", "ZonedDateTime", "Instant");
System.out.printf("%-35s %s%n", theZDT, theZDT.toInstant());
System.out.printf("%-35s %s%n", zdtSameInstant, zdtSameInstant.toInstant());
```

```
          ZonedDateTime                   Instant
1977-08-16T09:30-10:00[US/Hawaii]   1977-08-16T19:30:00Z
1977-08-16T14:30-05:00[US/Central]  1977-08-16T19:30:00Z
```

```
ZonedDateTime withFIELD(int amount)
ZonedDateTime with(TemporalField field, long newValue)
ZonedDateTime withZoneSameLocal(ZoneId zone)
ZonedDateTime withZoneSameInstant(ZoneId zone)
```

```
import java.time.DateTimeException;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;

public class FlightTimeInfo {
  public static void main(String[] args) {
    try {
      // Departure from New York at 8:30pm on July 4, 2021.                  (1)
      LocalDateTime departure = LocalDateTime.of(2021, Month.JULY, 4, 20, 30);
      ZoneId departureZone = ZoneId.of("America/New_York");
      ZonedDateTime departureZDT = ZonedDateTime.of(departure, departureZone);

      // Flight time is 7 hours and 30 minutes.
      // Calculate local arrival time at London:                              (2)
      ZoneId arrivalZone = ZoneId.of("Europe/London");
      ZonedDateTime arrivalZDT
          = departureZDT.withZoneSameInstant(arrivalZone)
                        .plusMinutes(7*60 + 30);
      System.out.printf("DEPARTURE:  %s%n", departureZDT);
      System.out.printf("ARRIVAL:    %s%n", arrivalZDT);

      // Flight time as a Duration:                                           (3)
      Duration flightduration = Duration.between(departureZDT, arrivalZDT);
      System.out.println("Flight duration:     " + flightduration);
      // Flight time in minutes:                                             (4)
      long flightTime = departureZDT.until(arrivalZDT, ChronoUnit.MINUTES);
      System.out.println("Flight time (mins.): " + flightTime);

      System.out.printf(                                                  // (5)
          "Time at departure airport on arrival: %s%n",
          departureZDT.plusMinutes(7*60 + 30));

      System.out.printf(                                                  // (6)
          "Time at departure airport on arrival: %s%n",
          arrivalZDT.withZoneSameInstant(departureZone));

    } catch (DateTimeException e) {
      e.printStackTrace();
    }
  }
}
```

```
DEPARTURE:  2021-07-04T20:30-04:00[America/New_York]
ARRIVAL:    2021-07-05T09:00+01:00[Europe/London]
Flight duration:     PT7H30M
Flight time (mins.): 450
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
Time at departure airport on arrival: 2021-07-05T04:00-04:00[America/New_York]
```

```
ZonedDateTime plusUNIT(long amount)
ZonedDateTime minusUNIT(long amount)
ZonedDateTime plus(long amountToAdd, TemporalUnit unit)
ZonedDateTime minus(long amountToSubtract, TemporalUnit unit)
boolean isSupported(TemporalUnit unit)
```

```
ZonedDateTime plus(TemporalAmount amountToAdd)
ZonedDateTime minus(TemporalAmount amountToSubtract)
```

```
long until(Temporal endExclusive, TemporalUnit unit)
```

```
Daylight Savings in US/Central starts at 2021-03-14T02:00 (spring forward 1 hour).
                    ___________ZonedDateTime__________
                       Date    Time  Offset    TZ       DST   UTC
(1) Before gap:     2021-03-14 01:30 -06:00 US/Central false 07:30
    + 1 hour
(2) After gap:      2021-03-14 03:30 -05:00 US/Central true  08:30
```

```
Daylight Savings in US/Central ends at 2021-11-07T02:00 (fall back 1 hour).
                    ___________ZonedDateTime__________
                       Date    Time  Offset    TZ       DST   UTC
(1) Before overlap: 2021-11-07 00:30 -05:00 US/Central true  05:30
    + 1 hour
(2) In overlap:     2021-11-07 01:30 -05:00 US/Central true  06:30
    + 1 hour
(3) In overlap:     2021-11-07 01:30 -06:00 US/Central false 07:30
    + 1 hour
(4) After overlap:  2021-11-07 02:30 -06:00 US/Central false 08:30
```

```
// java.time.zone.ZoneRules
boolean isDaylightSavings(Instant instant)
```

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;

public class DSTAdjustment {
  public static void main(String[] args) {
    adjustForGap();
    adjustForOverlap();
  }

  /**
   * Adjustment due to the time gap at DST crossover.                    (1a)
   * DST starts in US/Central TZ: 2021-03-14T02:00:00,
   * clocks are moved forward 1 hour, resulting in a time gap of 1 hour.
   */
  static void adjustForGap() {
    // Start date and time for DST in US/Central in 2021.                (2a)
    ZoneId cTZ = ZoneId.of("US/Central");
    LocalDate dateStartDST = LocalDate.of(2021, 3, 14);
    LocalTime timeStartDST = LocalTime.of(2, 0);
    LocalDateTime ldtStartDST = LocalDateTime.of(dateStartDST, timeStartDST);
    ZonedDateTime zdtStartDST = ZonedDateTime.of(ldtStartDST, cTZ);

    // Time before the gap.                                                 (3a)
    LocalTime timeBeforeGap = LocalTime.of(1, 30);
    LocalDateTime ldtBeforeGap = LocalDateTime.of(dateStartDST, timeBeforeGap);
    ZonedDateTime zdtBeforeGap = ZonedDateTime.of(ldtBeforeGap, cTZ);

    // Add 1 hour.                                                          (4a)
    ZonedDateTime zdtAfterGap = zdtBeforeGap.plusHours(1);

    // Print a report.                                                      (5a)
    System.out.printf("Daylight Savings in %s starts at %s "
        + "(spring forward 1 hour).%n", cTZ, ldtStartDST);
    System.out.println("                    ___________ZonedDateTime__________");

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before gap:     ", zdtBeforeGap);
    System.out.println("    + 1 hour");
    printInfo("(2) After gap:      ", zdtAfterGap);
    System.out.println();

    // Add 3 hours:                                                         (6a)
    ZonedDateTime zdtPlus3Hrs = zdtBeforeGap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeGap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Adjustment due to the time overlap at DST crossover.                     (1b)
   * DST ends in US/Central TZ: 2021-11-07T02:00:00,
   * clocks are moved backward 1 hour, resulting in a time overlap of 1 hour.
   */
  static void adjustForOverlap() {
    // End date and time for DST in US/Central in 2021.                     (2b)
    ZoneId cTZ = ZoneId.of("US/Central");
    LocalDate dateEndDST = LocalDate.of(2021, 11, 7);
    LocalTime timeEndDST = LocalTime.of(2, 0);
    LocalDateTime ldtEndDST = LocalDateTime.of(dateEndDST, timeEndDST);
    ZonedDateTime zdtEndDST = ZonedDateTime.of(ldtEndDST, cTZ);

    // Time before the overlap:                                             (3b)
    LocalTime timeBeforeOverlap = LocalTime.of(0, 30);
    LocalDateTime ldtBeforeOverlap = LocalDateTime.of(dateEndDST,
                                                      timeBeforeOverlap);
    ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(ldtBeforeOverlap, cTZ);

    // Add 1 hour:                                                          (4b)
    ZonedDateTime zdtInOverlap1 = zdtBeforeOverlap.plusHours(1);
    ZonedDateTime zdtInOverlap2 = zdtInOverlap1.plusHours(1);
    ZonedDateTime zdtAfterOverlap = zdtInOverlap2.plusHours(1);
    // Print a report.                                                       (5b)
    System.out.printf("Daylight Savings in %s ends at %s (fall back 1 hour).%n",
                      cTZ, ldtEndDST);
    System.out.println("                    ___________ZonedDateTime__________");

    System.out.printf("%27s %7s %7s %5s %9s %5s%n",
                      "Date", "Time", "Offset", "TZ", "DST", "UTC");
    printInfo("(1) Before overlap: ", zdtBeforeOverlap);
    System.out.println("    + 1 hour");
    printInfo("(2) In overlap:     ", zdtInOverlap1);
    System.out.println("    + 1 hour");
    printInfo("(3) In overlap:     ", zdtInOverlap2);
    System.out.println("    + 1 hour");
    printInfo("(4) After overlap:  ", zdtAfterOverlap);
    System.out.println();

    // Add 3 hours:                                                          (6b)
    ZonedDateTime zdtPlus3Hrs = zdtBeforeOverlap.plusHours(3);
    System.out.printf("%s + 3 hours = %s%n", zdtBeforeOverlap, zdtPlus3Hrs);
    System.out.println();
  }

  /**
   * Print info for a date-time.                                             (7)
   * @param leadTxt Text to lead the information.
   * @param zdt     Zoned date-time whose info is printed.
   */
  static void printInfo(String leadTxt, ZonedDateTime zdt) {
    System.out.printf(leadTxt + "%10s %5s %6s %5s %-5s %5s%n",
                      zdt.toLocalDate(), zdt.toLocalTime(),
                      zdt.getOffset(), zdt.getZone(),
                      isDST(zdt), localTimeAtUTC(zdt));
  }

  /**
   * Determine if DST is in effect for a zoned date-time.                    (8)
   * @param zdt  Zoned date-time whose DST status should be determined.
   * @return     true, if DST is in effect.
   */
  static boolean isDST(ZonedDateTime zdt) {
    return zdt.getZone().getRules().isDaylightSavings(zdt.toInstant());
  }

  /**
   *  Find local time at UTC/Greenwich equivalent to local time in           (9)
   *  the specified zoned date-time.
   * @param zdt   Zoned date-time to convert to UTC/Greenwich.
   * @return      Equivalent local time at UTC/Greenwich.
   */
  static LocalTime localTimeAtUTC(ZonedDateTime zdt) {
    return zdt.withZoneSameInstant(ZoneOffset.UTC).toLocalTime();
  }
}
```

```
ZonedDateTime zdt1 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14),
    LocalTime.of(2, 30),                 // Time 02:30 is in the gap.
    ZoneId.of("US/Central"));
System.out.println(zdt1);                // 2021-03-14T03:30-05:00[US/Central]
```

```
ZonedDateTime zdt2 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(1, 30),                // Time 01:30 in the overlap.
    ZoneId.of("US/Central"));
System.out.println(zdt2);               // 2021-11-07T01:30-05:00[US/Central]
```

```
ZonedDateTime zdt3 = ZonedDateTime.of(
    LocalDate.of(2021, 3, 14), LocalTime.of(0, 0), ZoneId.of("US/Central")
  ).withHour(2);                      // Time 02:00 is in the gap.
System.out.println(zdt3);             // 2021-03-14T03:00-05:00[US/Central]
```

```
ZonedDateTime zdt4 = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7), LocalTime.of(1, 0), ZoneId.of("US/Central")
  ).withMinute(30);                 // Time 01:30 is in the overlap.
System.out.println(zdt4);           // 2021-11-07T01:30-05:00[US/Central]
```

```
ZonedDateTime zdtBeforeOverlap = ZonedDateTime.of(
    LocalDate.of(2021, 11, 7),
    LocalTime.of(0, 30),              // Time 00:30 is before the DST crossover.
    ZoneId.of("US/Central"));
```

```
// Date units and time units.
System.out.printf("%s + 1 day    = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusDays(1));        // (1) Add 1 day.
System.out.printf("%s + 24 hours = %s%n",
    zdtBeforeOverlap, zdtBeforeOverlap.plusHours(24));      // (2) Add 24 hours.
```

```
2021-11-07T00:30-05:00[US/Central] + 1 day    = 2019-11-04T00:30-06:00[US/Central]
2021-11-07T00:30-05:00[US/Central] + 24 hours = 2021-11-07T23:30-06:00[US/Central]
```

```
import java.time.*;
import java.util.Date;

public class ConvertToLegacyDate {
  /** Convert a ZonedDateTime to Date. */
  public static Date zdtToDate(ZonedDateTime zdt) {                        // (1)
    return Date.from(zdt.toInstant());
  }

  /** Convert a LocalDateTime to Date. */
  public static Date ldtToDate(LocalDateTime ldt) {                        // (2)
    return Date.from(ldt.atZone(ZoneId.systemDefault()).toInstant());
  }

  /** Convert a LocalDate to Date. */
  public static Date ldToDate(LocalDate ld) {                              // (3)
    return Date.from(ld.atStartOfDay()
                       .atZone(ZoneId.systemDefault())
                       .toInstant());
  }

  /** Convert a LocalTime to Date. */
  public static Date ltToDate(LocalTime lt) {                              // (4)
    return Date.from(lt.atDate(LocalDate.of(2021, 1, 1))
                       .atZone(ZoneId.systemDefault())
                       .toInstant());
  }
}
```

```
import java.time.LocalDate;

public class RQ1 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 31);
    LocalDate d2 = d1.plusMonths(1);
    LocalDate d3 = d2.minusMonths(1);
    System.out.println(d1.getDayOfYear() + " " + d2.getDayOfYear() + " " +
                       d3.getDayOfYear());
  }
}
```

```
import java.time.LocalDate;

public class RQ2 {
  public static void main(String[] args) {
    LocalDate d1 = LocalDate.of(2021, 1, 1);
    d1 = d1.withDayOfMonth(d1.lengthOfMonth()).withMonth(2);
    System.out.println(d1);
  }
}
```

```
import java.time.*;

public class RQ3 {
  public static void main(String[] args) {
    LocalDateTime d1 = LocalDate.of(2021, 4, 1).atStartOfDay();
    Instant i1 = d1.toInstant(ZoneOffset.of("+18:00"));
    LocalDate d2 = LocalDate.ofInstant(i1, ZoneId.of("UTC"));
    System.out.println(d2);
  }
}
```

```
import java.time.*;

public class RQ4 {
  public static void main(String[] args) {
    LocalDateTime dt = LocalDate.of(2021, 4, 1).atStartOfDay();
    ZonedDateTime zdt1 = dt.atZone(ZoneId.of("Europe/Paris"));
    ZonedDateTime zdt2 = dt.atZone(ZoneId.of("Europe/London"));
    Duration d = Duration.between(zdt1.minusMinutes(30), zdt2.plusMinutes(30));
    System.out.println(d);
  }
}
```

```
import java.time.*;

public class RQ6 {
  public static void main(String[] args) {
    LocalTime t = LocalTime.of(8, 15);
    LocalDate d = LocalDate.of(2021, 4, 1);
    LocalDateTime dt = d.atTime(t);
    dt.minusMinutes(30).withDayOfMonth(12);
    System.out.println(dt);
  }
}
```

```
import java.time.*;

public class RQ8 {
  public static void main(String[] args) {
    Period d  = Period.parse("P2D");
    LocalDate ld = LocalDate.of(2021, 4, 1);
    LocalDateTime ldt = ld.plus(d);
    System.out.println(ldt);
  }
}
```

```
import java.time.*;

public class RQ9 {
  public static void main(String[] args) {
    Duration d  = Duration.parse("PT-24H");
    LocalDate ld = LocalDate.of(2021, 4, 1).plus(d);
    System.out.println(ld);
  }
}
```

```
import java.time.*;

public class RQ10 {
  public static void main(String[] args) {
    LocalDate ld = LocalDate.of(2021, 4, 1);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

```
LocalDateTime ldt = ld.atTime(LocalTime.of(0, 30))
                      .plus(Duration.ofHours(48));
```

```
LocalDateTime ldt = ld.plus(Duration.ofHours(48))
                      .atTime(LocalTime.of(0, 30));
```

```
LocalDateTime ldt = ld.atTime(LocalTime.of(48,30));
LocalDateTime ldt = ld.plusDays(3).atTime(LocalTime.of(-23,30));
LocalDateTime ldt = ld.atTime(0, 30).plus(Duration.ofHours(48));
```

```
import java.time.*;

public class RQ11 {
  public static void main(String[] args) {
    LocalDate now = LocalDate.now();
    LocalDate foolsDay = LocalDate.of(2021, Month.APRIL, 1);
    LocalDateTime tomorrowAfternoon = now.plusDays(1)
                                         .atTime(LocalTime.of(12, 01));
    LocalDate anotherDay = foolsDay.withDayOfMonth(2).minusDays(1);
  }
}
```

```
import java.time.*;

public class RQ13 {
  public static void main(String[] args) {
    LocalTime lt = LocalTime.of(17,30);
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    Duration d = Duration.between(lt, ldt);
    System.out.println(d);
  }
}
```

```
import java.time.*;
import static java.time.temporal.ChronoUnit.DAYS;

public class RQ14 {
  public static void main(String[] args) {
    LocalDateTime ldt = LocalDateTime.of(2021, Month.APRIL, 2, 15, 15);
    // (1) INSERT CODE HERE
    System.out.println(ldt);
  }
}
```

```
ldt = ldt.plusHours(1).with(LocalDate.of(2021, Month.APRIL, 3));
ldt = ldt.plusDays(1).with(LocalTime.of(16, 15));
ldt = ldt.plus(Duration.of(2, DAYS)).minus(Duration.parse("PT23H"));
ldt = ldt.plus(Duration.of(2, DAYS))
          .minus(Duration.ofMinutes(15).ofHours(16));
```

```
ldt = ldt.plus(Duration.parse("PT25H"));
ldt = ldt.plus(Duration.ofMinutes(25 * 60));
```

```
Locale(String language)
Locale(String language, String country)
```

```
static Locale getDefault()
static void   setDefault(Locale newLocale)
static Locale[] getAvailableLocales()
static Locale forLanguageTag(String languageTag)
```

```
String getDisplayCountry()
String getDisplayCountry(Locale inLocale)
String getDisplayLanguage()
String getDisplayLanguage(Locale inLocale)
String getDisplayName()
String getDisplayName(Locale inLocale)
```

```
import java.util.Locale;
public class LocalesEverywhere {

  public static void main(String[] args) {

    Locale locDF = Locale.getDefault();              // (1)
    Locale locNO =  new Locale("no", "NO");          // Locale: Norwegian/Norway
    Locale locFR =  new Locale("fr", "FR");          // Locale: French/France

    System.out.println("Default locale is: " + locDF.getDisplayName());
    System.out.println("Display country (language) for Norwegian locale:");

    System.out.printf("In %s: %s (%s)%n", locDF.getDisplayCountry(),
        locNO.getDisplayCountry(locDF), locNO.getDisplayLanguage(locDF));

    System.out.printf("In %s: %s (%s)%n", locNO.getDisplayCountry(),
        locNO.getDisplayCountry(locNO), locNO.getDisplayLanguage(locNO));

    System.out.printf("In %s: %s (%s)%n", locFR.getDisplayCountry(),
        locNO.getDisplayCountry(locFR), locNO.getDisplayLanguage(locFR));

    System.out.println("\nChanging the default locale.");
    Locale.setDefault(Locale.GERMANY);               // (2) Locale: German/Germany
    locDF = Locale.getDefault();
    System.out.println("Default locale is: " + locDF.getDisplayName());
    System.out.printf("Interpreting %s locale information in %s locale.%n",
                      locNO.getDisplayName(), locDF.getDisplayName());
  }
}
```

```
Default locale is: English (United States)
Display country (language) for Norwegian locale:
In United States: Norway (Norwegian)
In Norway: Norge (norsk)
In France: Norvège (norvégien)

Changing the default locale.
Default locale is: Deutsch (Deutschland)
Interpreting Norwegisch (Norwegen) locale information in Deutsch (Deutschland)
locale.
```

```
cert.name = OCP, Java SE 17 Developer
exam.title = Java SE 17 Developer
exam.number = 1Z0-829
```

```
company = GLOBUS
greeting=Hi!
        gratitude              =Thank you!
```

"Au revoir!":

```
farewell = Au \
           revoir!
```

```
baseName_languageCode_countryCode
```

`BasicResources`	Default resource bundle
`BasicResources_no_NO`	Resource bundle for Norwegian (Norway) locale
`BasicResources_fr`	Resource bundle for all locales with French language

```
# File: BasicResources.properties
company = GLOBUS
greeting = Hi!
gratitude = Thank you!
farewell = See you!
```

```
# File: BasicResources_no_NO.properties
greeting = Hei!
gratitude = Takk!
farewell = Ha det!
```

```
# File: BasicResources_fr.properties
greeting = Bonjour!
gratitude = Merci!
farewell = Au revoir!
```

```
static ResourceBundle getBundle(String baseName)
static ResourceBundle getBundle(String baseName, Locale locale)
```

```
Object getObject(String key)
String getString(String key)
```

```
import java.util.Locale;
import java.util.ResourceBundle;
public class UsingResourceBundles {

  // Supported locales:                                                    // (1)
  public static final Locale[] locales = {
    Locale.getDefault(),                                 // Default: US (English)
    new Locale("no", "NO"),                              // Norway (Norwegian)
    Locale.FRANCE,                                       // France (French)
    Locale.CANADA_FRENCH                                 // Canada (French)
  };

  // Localized data from property resource files:                    // (2)
  private static String company;
  private static String greeting;
  private static String gratitude;
  private static String farewell;

  public static void main(String[] args) {                           // (3)
    for (Locale locale : locales) {
      setLocaleSpecificData(locale);
      printLocaleSpecificData(locale);
    }
  }

  private static void setLocaleSpecificData(Locale locale) {         // (4)
    // Get resources from property resource files:
    ResourceBundle properties =
        ResourceBundle.getBundle("resources.BasicResources", locale);     // (5)
    company = properties.getString("company");                       // (6)
    greeting = properties.getString("greeting");
    gratitude = properties.getString("gratitude");
    farewell = properties.getString("farewell");
  }

  private static void printLocaleSpecificData(Locale locale) {       // (7)
    System.out.println("Resources for " + locale.getDisplayName() + " locale:");
    System.out.println("Company: " + company);
    System.out.println("Greeting: " + greeting);
    System.out.println("Gratitude: " + gratitude);
    System.out.println("Farewell: " + farewell);
    System.out.println();
  }
}
```

```
Resources for English (United States) locale:
Company: GLOBUS
Greeting: Hi!
Gratitude: Thank you!
Farewell: See you!

Resources for Norwegian (Norway) locale:
Company: GLOBUS
Greeting: Hei!
Gratitude: Takk!
Farewell: Ha det!

Resources for French (France) locale:
Company: GLOBUS
Greeting: Bonjour!
```

```
Gratitude: Merci!
Farewell: Au revoir!

Resources for French (Canada) locale:
Company: GLOBUS
Greeting: Bonjour!
Gratitude: Merci!
Farewell: Au revoir!
```

```
String company;
...
ResourceBundle properties =
    ResourceBundle.getBundle("resources.BasicResources", locale);    // (5)
company = properties.getString("company");                           // (6)
```

```
BasicResources.properties
BasicResources_no_NO.properties
BasicResources_fr.properties
```

```
ResourceBundle resources = ResourceBundle.getBundle(baseName, specifiedLocale);
```

```
baseName_specifiedLanguageCode_specifiedCountryCode
baseName_specifiedLanguageCode
```

```
baseName_defaultLanguageCode_defaultCountryCode
baseName_defaultLanguageCode
```

```
BasicResources_fr.properties
BasicResources.properties (parent)
```

```
# MyResources_fr_CA.properties
file1 = MyResources_fr_CA
```

```
# MyResources_fr.properties
file2 = MyResources_fr
```

```
# MyResources_en_US.properties
file3 = MyResources_en_US
```

```
# MyResources_en.properties
file4 = MyResources_en
```

```
# MyResources.properties
file5 = MyResources
```

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.TreeSet;

public class LocatingBundles {
  public static void main(String[] args) {

    Locale[] locales = {
        new Locale("no", "NO"),                               // Norway
        Locale.CANADA_FRENCH,                                 // Canada (French)
        Locale.FRENCH,                                        // French
        Locale.getDefault(),                                  // Default: en_US
    };

    for (Locale locale: locales) {
      System.out.println("Locating resource bundles for " + locale + " locale:");
      ResourceBundle resources = ResourceBundle.getBundle("resources.MyResources",
                                                          locale);
      for (String key : new TreeSet<>(resources.keySet())) {                // (1)
        System.out.println(resources.getString(key));
      }
```

```
            System.out.println();
        }
    }
}
```

```
Locating resource bundles for no_NO locale:
MyResources_en_US
MyResources_en
MyResources

Locating resource bundles for fr_CA locale:
MyResources_fr_CA
MyResources_fr
MyResources

Locating resource bundles for fr locale:
MyResources_fr
MyResources

Locating resource bundles for en_US locale:
MyResources_en_US
MyResources_en
MyResources
```

```
# File: MyResources_en_US.properties
greeting = Howdy!
gratitude = Thank you!
farewell = See ya!
farewell = Bye!
```

```
import java.util.*;
public class TestResourceBundles {
  public static void main(String[] args) {
    ResourceBundle resources = ResourceBundle.getBundle("pkg.MyResources",
                                                         Locale.FRANCE);
    for (String key : resources.keySet()) {
      System.out.println(resources.getString(key));
    }
  }
}
```

```
# MyResources_en_US.properties
farewell = See ya!
```

```
# MyResources_en.properties
farewell = Have a good one!
```

```
# MyResources.properties
farewell = Goodbye!
```

```
import java.util.*;
public class TestResourceBundles3 {
  public static void main(String[] args) {
    ResourceBundle resources = ResourceBundle.getBundle("pkg3.MyResources",
                                                         Locale.ENGLISH);
    System.out.println(resources.getString("farewell"));
  }
}
```

```
import java.util.*;

public class RQ4_2 {
  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("resources", new Locale("en"));
    for (String key : new TreeSet<String>(rb.keySet())) {
      System.out.print(key+"="+rb.getString(key)+" ");
    }
  }
}
```

```
import java.util.*;

public class RQ4_3 {
  public static void main(String[] args) {
    Locale.setDefault(new Locale("ru"));
    ResourceBundle rb = ResourceBundle.getBundle("resources",new Locale("en"));
    for (String key : new TreeSet<String>(rb.keySet())) {
      System.out.print(key+"="+rb.getString(key)+" ");
    }
  }
}
```

```
static NumberFormat getNumberInstance()
static NumberFormat getNumberInstance(Locale locale)
static NumberFormat getCurrencyInstance()
static NumberFormat getCurrencyInstance(Locale locale)
static NumberFormat getPercentInstance()
static NumberFormat getPercentInstance(Locale locale)
static NumberFormat getCompactNumberInstance()
static NumberFormat getCompactNumberInstance(Locale locale,
                                             NumberFormat.Style formatStyle)
```

```
String format(Object obj)              // inherited from the Format class.
```

```
double num = 12345.6789;                                // (1a)
// BigDecimal num = new BigDecimal("12345.6789");       // (1b)

Locale locNOR = new Locale("no", "NO");                 // Norway
NumberFormat nfNOR = NumberFormat.getNumberInstance(locNOR);
System.out.println(nfNOR.format(num));                  // 12 345,679

NumberFormat nfUS = NumberFormat.getNumberInstance(Locale.US);
System.out.println(nfUS.format(num));                   // 12,345.679
```

```
NumberFormat cfNOR = NumberFormat.getCurrencyInstance(locNOR);
String formattedCurrStr = cfNOR.format(num);
System.out.println(formattedCurrStr);              // kr 12 345,68 (with 2 nbsp)

NumberFormat cfUS = NumberFormat.getCurrencyInstance(Locale.US);
String formattedCurrStrUS = cfUS.format(num);
System.out.println(formattedCurrStrUS);            // $12,345.68
```

```
double rebate = 0.746;
NumberFormat pfNOR = NumberFormat.getPercentInstance(locNOR);
String formattedPStr = pfNOR.format(rebate);
System.out.println(formattedPStr);                // 75 %  (with nbsp)

NumberFormat pfUS = NumberFormat.getPercentInstance(Locale.US);
String formattedPStrUS = pfUS.format(rebate);
System.out.println(formattedPStrUS);              // 75%

System.out.println(pfUS.format(0.745));           // 74%
```

```
NumberFormat df0 = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(df0.format(-9.99));            // -$9.99
```

```
Locale loc = Locale.forLanguageTag("en-US-u-cf-account");   // (1)
NumberFormat df = NumberFormat.getCurrencyInstance(loc);    // (2)
System.out.println(df.format(-9.99));                       // ($9.99)
```

```
Number parse(String str) throws ParseException
```

```
void setParseIntegerOnly(boolean intOnly)
boolean isParseIntegerOnly()
```

```
void setParseBigDecimal(boolean newValue)
boolean isParseBigDecimal()
```

```
System.out.println(nfNOR.parse("9876.598"));     // (1) 9876
System.out.println(nfNOR.parse("9876,598"));     // (2) 9876.598

System.out.println(nfUS.parse("9876.598"));      // (3) 9876.598
System.out.println(nfUs.parse("9876,598"));      // (4) 9876598
```

```
DecimalFormat dfUS = (DecimalFormat) nfUS;
dfUS.setParseBigDecimal(true);
BigDecimal bd = (BigDecimal) dfUS.parse("9876,598");
```

```
System.out.println(cfNOR.parse("kr\u00a09876.59"));          // (1) 9876
System.out.println(cfNOR.parse("kr\u00a09876,59"));          // (2) 9876.59
System.out.println(cfNOR.parse("kr\u00a09 876,59"));         // (3) 9
System.out.println(cfNOR.parse("kr\u00a09\u00a0876,59"));    // (4) 9876.59

System.out.println(cfUS.parse("$9876.59"));                  // (5) 9876.59
System.out.println(pfNOR.parse("15,75\u00a0%"));              // (1) 0.1575
System.out.println(pfUS.parse("25.5%"));                      // (2) 0.255
```

```
NumberFormat shortCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.SHORT)
NumberFormat longCompactFormat = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.LONG);
```

```
System.out.println(shortCompactFormat.format(9_400_000));    // 9M
System.out.println(longCompactFormat.format(9_400_000));     // 9 million
```

```
shortCompactFormat.setMaximumFractionDigits(2);
longCompactFormat.setMaximumFractionDigits(2);
```

```
try {
  System.out.println(shortCompactFormat.parse("9M"));         // 9000000
  System.out.println(longCompactFormat.parse("9 million"));  // 9000000
} catch (ParseException pe) {
  System.out.println(pe);
}
```

```
void setMinimumIntegerDigits(int n)
int  getMinimumIntegerDigits()

void setMaximumIntegerDigits(int n)
int  getMaximumIntegerDigits()

void setMinimumFractionDigits(int n)
int  getMinimumFractionDigits()

void setMaximumFractionDigits(int n)
int  getMaximumFractionDigits()
```

```
void setRoundingMode(RoundingMode roundingMode)
```

```
 1.5 ->  2 (same as UP, if discarded fraction is >= 0.5)
 1.4 ->  1 (same as DOWN, if discarded fraction is < 0.5)
-1.4 -> -1 (same as DOWN, if discarded fraction is < 0.5)
-1.5 -> -2 (same as UP, if discarded fraction is >= 0.5)
```

```
 1.6 ->  2 (same as UP, if discarded fraction is > 0.5)
 1.5 ->  1 (same as DOWN, if discarded fraction is <= 0.5)
-1.5 -> -1 (same as DOWN, if discarded fraction is <= 0.5)
-1.6 -> -2 (same as UP, if discarded fraction is > 0.5)
```

```
 2.5 ->  2 (same as HALF_DOWN, if digit left of discarded fraction is even)
 1.5 ->  2 (same as HALF_UP, if digit left of discarded fraction is odd)
-1.5 -> -2 (same as HALF_UP, if digit left of discarded fraction is odd)
-2.5 -> -2 (same as HALF_DOWN, if digit left of discarded fraction is even)
```

```
import java.math.RoundingMode;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class Rounding {

  public static void main(String[] args) {
    System.out.println(" Rounding:   v1          v2");
    NumberFormat nfmtUS = NumberFormat.getNumberInstance(Locale.US);     // (1)
    int maxFractionDigits = 0;                                           // (2)
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.CEILING,   1.1, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.FLOOR,     1.8, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.UP,        1.1, -1.1);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.DOWN,      1.8, -1.8);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   1.5, 1.4);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_UP,   -1.4, -1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, 1.6, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_DOWN, -1.5, -1.6);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, 2.5, 1.5);
    roundIt(nfmtUS, maxFractionDigits, RoundingMode.HALF_EVEN, -1.5, -2.5);
  }

  static void roundIt(NumberFormat nf, int maxFractionDigits, RoundingMode rMode,
                      double v1, double v2) {                    // (3)
    nf.setMaximumFractionDigits(maxFractionDigits);              // (4)
    nf.setRoundingMode(rMode);                                   // (5)
    System.out.printf("%9s: ", rMode);
    System.out.printf("%5s -> %2s ", v1, nf.format(v1));         // (6)
    System.out.printf("%5s -> %2s%n", v2, nf.format(v2));        // (7)
  }
}
```

```
 Rounding:   v1          v2
  CEILING:   1.1 ->  2  -1.8 -> -1
    FLOOR:   1.8 ->  1  -1.1 -> -2
       UP:   1.1 ->  2  -1.1 -> -2
     DOWN:   1.8 ->  1  -1.8 -> -1
  HALF_UP:   1.5 ->  2   1.4 ->  1
  HALF_UP:  -1.4 -> -1  -1.5 -> -2
HALF_DOWN:   1.6 ->  2   1.5 ->  1
HALF_DOWN:  -1.5 -> -1  -1.6 -> -2
HALF_EVEN:   2.5 ->  2   1.5 ->  2
HALF_EVEN:  -1.5 -> -2  -2.5 -> -2
```

```
NumberFormat df = NumberFormat.getNumberInstance(Locale.US);
if (nf intanceof DecimalFormat) {
  DecimalFormat df = (DecimalFormat) nf;
  df.applyPattern(pattern);                // Supply the pattern.
  String output = df.format(number);       // Format the number.
}
```

```
void applyPattern(String pattern)
```

```
import java.math.BigDecimal;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

public class FormattingDecimalNumbers {

  public static void main(String[] args) {
    // The number to format.
    double number = 1234.567;                                          // (1a)
//  BigDecimal number = new BigDecimal("1234.567");                    // (1b)

    // Formats to use:
    String[] patterns = {                                              // (2)
        "#",
        "###,###.##",
        "###.##",
        "00000.00",
        "BTC ###,###.##",                                       // BTC: Bitcoin
    };

    // Locales to consider:
    Locale[] locales = { Locale.US, Locale.GERMANY };                 // (3)

    // Create localized DecimalFormats:                                   (4)
    List<DecimalFormat> dfs = new ArrayList<>(locales.length);
    for (int i = 0; i < locales.length; i++) {
      NumberFormat nf = NumberFormat.getNumberInstance(locales[i]);
      if (nf instanceof DecimalFormat) {
        dfs.add((DecimalFormat) nf);
      }
    }

    // Write the header:
    System.out.printf("%15s", "Patterns");
    for (Locale locale : locales) {
      System.out.printf("%15s", locale);
    }
    System.out.println();

    // Do formatting and print results:
    for (String pattern : patterns) {
      System.out.printf("%15s", pattern);
      for (DecimalFormat df : dfs) {
        df.applyPattern(pattern);                                    // (5)
        String output = df.format(number);                            // (6)
        System.out.printf("%15s", output);
      }
      System.out.println();
    }
  }
}
```

```
      Patterns          en_US          de_DE
             #           1235           1235
    ###,###.##       1,234.57       1.234,57
        ###.##        1234.57        1234,57
      00000.00       01234.57       01234,57
BTC ###,###.##   BTC 1,234.57   BTC 1.234,57
```

```
String resultStr1 = date.format(df);
String resultStr2 = df.format(date);
boolean eqStr = resultStr1.equals(resultStr2);    // true
```

```
// Defined by LocalTime, LocalDate, LocalDateTime, and ZonedDateTime
String format(DateTimeFormatter formatter)
```

```
static TemporalType parse(CharSequence text)
static TemporalType parse(CharSequence text, DateTimeFormatter formatter)
```

```
// Defined by DateTimeFormatter
DateTimeFormatter localizedBy(Locale locale)
```

```
String format(TemporalAccessor temporal)
```

```
TemporalAccessor parse(CharSequence text)
```

```
DateTimeFormatter withZone(ZoneId zone)
```

```
LocalTime time = LocalTime.of(12, 30, 15, 99);
String strTime = time.toString();                  // (1) 12:30:15.000000099
LocalTime parsedTime = LocalTime.parse(strTime);  // (2)
System.out.println(time.toString().equals(parsedTime.toString())); // true
```

```
LocalTime badTime = LocalTime.parse("12.30.15");  // DateTimeParseException
```

```
DateTimeFormatter df = DateTimeFormatter.ISO_LOCAL_DATE;     // (1)
LocalDate date = LocalDate.of(1935, 1, 8);
String strDate = date.format(df);                            // (2) 1935-01-08
LocalDate parsedDate = LocalDate.parse(strDate, df);         // (3)
System.out.println(strDate + "|" +
                   parsedDate.format(df));          // (4) 1935-01-08|1935-01-08
```

```
LocalDateTime dateTime = LocalDateTime.of(1935, 1, 8, 12, 45);
String strDate2 = dateTime.format(df);                         // (4) 1935-01-08
LocalDate parsedDate2 = LocalDate.parse(strDate2, df);       // (5) LocalDate
```

```
String timeStr2 = LocalTime.NOON.format(df);    // UnsupportedTemporalTypeException
LocalTime time2 = LocalTime.parse("12:00", df);// DateTimeParseException
```

```
static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)
static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle,
                                             FormatStylse timeStyle)
```

```
LocalTime time = LocalTime.of(14, 15, 30);             // 14:15:30
LocalDate date = LocalDate.of(2021, 12, 1);            // 2021-12-01
LocalDateTime dateTime = LocalDateTime.of(date, time); // 2021-12-01T14:15:30
// 2021-12-01T14:15:30-06:00[US/Central]
ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, ZoneId.of("US/Central"));
```

```
DateTimeFormatter dfs = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
String str1 = date.format(dfs);           // 12/1/21
String str2 = dateTime.format(dfs);       // Date part: 12/1/21
String str3 = zonedDateTime.format(dfs);  // Date part: 12/1/21
```

```
DateTimeFormatter tff = DateTimeFormatter.ofLocalizedTime(FormatStyle.FULL);
String str4 = zonedDateTime.format(tff);  // Time part:
                                          //   2:15:30 PM Central Standard Time
String str5 = time.format(tff);           // java.time.DateTimeException
String str6 = date.format(tff);           // java.time.temporal.
                                          //      UnsupportedTemporalTypeException
String str7 = dateTime.format(tff);       // java.time.DateTimeException
```

```
DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);// (1)
String inputStr = "2/29/21";                          // (2) en_US date, SHORT style
LocalDate parsedDate = LocalDate.parse(inputStr, df);    // (3)
System.out.println(parsedDate);                          // (4) 2021-02-28
```

```
DateTimeFormatter dtff
    = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL)
                       .localizedBy(Locale.FRANCE);          // (1)

// "mercredi 1 décembre 2021 à 14:15:30 heure normale du centre nord-américain"
String charSeq = zonedDateTime.format(dtff);                 // (2)
LocalTime pTime = LocalTime.parse(charSeq, dtff);            // (3) 14:15:30
LocalDate pDate = LocalDate.parse(charSeq, dtff);            // (4) 2021-12-01

// 2021-12-01T14:15:30
LocalDateTime pDateTime = LocalDateTime.parse(charSeq, dtff);         // (5)

// 2021-12-01T14:15:30-06:00[America/Chicago]
ZonedDateTime pZonedDateTime = ZonedDateTime.parse(charSeq, dtff);    // (6)
```

```
static DateTimeFormatter ofPattern(String pattern)
```

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;

public class FormattingParsingWithPatterns {

  /** Temporals */
  private static LocalTime time = LocalTime.of(12, 30, 15, 99);
  private static LocalDate date = LocalDate.of(2021, 4, 28);
  private static LocalDateTime dateTime = LocalDateTime.of(date, time);
  private static ZoneId zID = ZoneId.of("US/Central");
  private static ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, zID);

  public static void main(String[] args) {                                  // (1)
    Locale locale = Locale.US;
    usingTimePattern(locale);
    usingDatePattern(locale);
    usingDateTimePattern(locale);
    usingZonedDateTimePattern(locale);
  }

  /** Pattern with time part. */
  public static void usingTimePattern(Locale locale) {                      // (2)
    String timePattern = "HH::mm::ss:SSS";
    DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern(timePattern)
                                                       .localizedBy(locale);
    String strTime = time.format(timeFormatter);
    LocalTime parsedTime = LocalTime.parse(strTime, timeFormatter);
    String strTime2 = dateTime.format(timeFormatter);
    String strTime3 = zonedDateTime.format(timeFormatter);

    System.out.printf("Time pattern: %s%n", timePattern);
    System.out.printf("LocalTime (formatted): %s%n", strTime);
    System.out.printf("LocalTime (parsed):    %s%n", parsedTime);
    System.out.printf("LocalDateTime (formatted time part): %s%n", strTime2);
    System.out.printf("ZonedDateTime (formatted time part): %s%n%n", strTime3);
  }

  /** Pattern with date part. */
  public static void usingDatePattern(Locale locale) {                      // (3)
    String datePattern = "EEEE, uuuu/MMMM/dd";
    DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern(datePattern)
                                                       .localizedBy(locale);
    String strDate = date.format(dateFormatter);
    LocalDate parsedDate = LocalDate.parse(strDate, dateFormatter);
    String strDate2 = dateTime.format(dateFormatter);
    String strDate3 = zonedDateTime.format(dateFormatter);
    System.out.printf("Date pattern: %s%n", datePattern);
    System.out.printf("LocalDate (formatted): %s%n", strDate);
    System.out.printf("LocalDate (parsed)   : %s%n", parsedDate);
    System.out.printf("LocalDateTime (formatted date part): %s%n", strDate2);
    System.out.printf("ZonedDateTime (formatted date part): %s%n%n", strDate3);
  }

  /** Pattern with date and time parts. */
  public static void usingDateTimePattern(Locale locale) {                  // (4)
    String dtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd";
    DateTimeFormatter dtFormatter = DateTimeFormatter.ofPattern(dtPattern)
                                                     .localizedBy(locale);
    String strDateTime = dateTime.format(dtFormatter);
    LocalDateTime parsedDateTime = LocalDateTime.parse(strDateTime,
                                                       dtFormatter);
    LocalDate parsedDate3 = LocalDate.parse(strDateTime, dtFormatter);
    LocalTime parsedTime3 = LocalTime.parse(strDateTime, dtFormatter);

    System.out.printf("DateTime pattern: %s%n", dtPattern);
    System.out.printf("LocalDateTime (formatted):    %s%n", strDateTime);
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate3);
    System.out.printf("LocalTime (parsed time part): %s%n%n", parsedTime3);
  }

  /** Pattern with time zone, date and time parts. */
  public static void usingZonedDateTimePattern(Locale locale) {             // (5)
    String zdtPattern = "EE, HH::mm::ss 'on' uuuu/MM/dd VV";
    DateTimeFormatter zdtFormatter = DateTimeFormatter.ofPattern(zdtPattern)
                                                      .localizedBy(locale);
    String strZonedDateTime = zonedDateTime.format(zdtFormatter);
    ZonedDateTime parsedZonedDateTime
        = ZonedDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDateTime parsedDateTime2
        = LocalDateTime.parse(strZonedDateTime, zdtFormatter);
    LocalDate parsedDate4 = LocalDate.parse(strZonedDateTime,
                                            zdtFormatter);
    LocalTime parsedTime4 = LocalTime.parse(strZonedDateTime,
                                            zdtFormatter);

    System.out.printf("ZonedDateTime pattern: %s%n", zdtPattern);
    System.out.printf("ZonedDateTime (formatted):    %s%n", strZonedDateTime);
    System.out.printf("ZonedDateTime (parsed):       %s%n", parsedZonedDateTime);
    System.out.printf("LocalDateTime (parsed):       %s%n", parsedDateTime2);
    System.out.printf("LocalDate (parsed date part): %s%n", parsedDate4);
    System.out.printf("LocalTime (parsed time part): %s%n", parsedTime4);
  }
}
```

```
Time pattern: HH::mm::ss:SSS
LocalTime (formatted): 12::30::15:000
LocalTime (parsed):    12:30:15
LocalDateTime (formatted time part): 12::30::15:000
ZonedDateTime (formatted time part): 12::30::15:000
```

```
Date pattern: EEEE, uuuu/MMMM/dd
LocalDate (formatted): Wednesday, 2021/April/28
LocalDate (parsed)   : 2021-04-28
LocalDateTime (formatted date part): Wednesday, 2021/April/28
ZonedDateTime (formatted date part): Wednesday, 2021/April/28

DateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd
LocalDateTime (formatted):    Wed, 12::30::15 on 2021/04/28
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15

ZonedDateTime pattern: EE, HH::mm::ss 'on' uuuu/MM/dd VV
ZonedDateTime (formatted):    Wed, 12::30::15 on 2021/04/28 US/Central
ZonedDateTime (parsed):       2021-04-28T12:30:15-05:00[US/Central]
LocalDateTime (parsed):       2021-04-28T12:30:15
LocalDate (parsed date part): 2021-04-28
LocalTime (parsed time part): 12:30:15
```

```
String pattern = "At {3} on {2} Elvis landed at {0} and was greeted by {1} fans.";
String output = MessageFormat.format(pattern, "Honolulu", 3000,
                              LocalDate.of(1961,3,25), LocalTime.of(12,15));
System.out.println(output);
```

At 12:15 on 1961-03-25 Elvis landed at Honolulu and was greeted by 3,000 fans.

```
Format element:     {0}      {1}           {2}                     {3}
Arguments:       "Honolulu", 3000, LocalDate.of(1961,3,25), LocalTime.of(12,15)
```

```
{ Argument_Index }
{ Argument_Index, Format_Type }
{ Argument_Index, Format_Type, Format_Style }
```

```
DateFormat.getTimeInstance(DateFormat.SHORT, getLocale())
```

```
// Specify the pattern:                                                    (1)
String pattern2 = "At {3,time,short} on {2,date,medium} Elvis landed at {0} "
               +  "and was greeted by {1,number,integer} fans.";

// Create a MessageFormat based on the given pattern:                      (2)
MessageFormat mf2 = new MessageFormat(pattern2);

// Create the array with the arguments to format:                          (3)
Object[] messageArguments = {
    "Honolulu",                                          // argument index 0
    3000,                                                // argument index 1
    ConvertToDate.ldToDate(LocalDate.of(1961,3,25)),    // argument index 2
    ConvertToDate.ltToDate(LocalTime.of(12,15))         // argument index 3
};
```

```
// Format the arguments:                                                  (4)
String output2 = mf2.format(messageArguments);
System.out.println(output2);
```

At 12:15 PM on Mar 25, 1961 Elvis landed at Honolulu and was greeted by 3,000 fans.

```
MessageFormat(String pattern)
MessageFormat(String pattern, Locale locale)
```

```
final String format(Object obj)       // Inherited from the Format class.
```

```
static String format(String pattern, Object... arguments)
```

```
void setFormat(int formatElementIndex, Format newFormat)
void setFormats(Format[] newFormats)
void setFormatByArgumentIndex(int argumentIndex, Format newFormat)
void setFormatsByArgumentIndex(Format[] newFormats)
```

```
# File: StockInfoBundle.properties
pattern = Stock date: {3,time,short}, {4,date,long}\n\
          Item name: {0}\n\
          Item price: {1,number,currency}\n\
          Number of items: {2,number,integer}
item.name = Frozen pizza
item.price = 9.99
```

```
import java.text.*;
import java.time.*;
import java.util.*;

public class CompoundMessageFormatting {
  static void displayStockInfo(Locale requestedLocale) {                  // (1)
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the relevant resource bundle:                                   (2)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.StockInfoBundle", requestedLocale);

    // Create a formatter, given the pattern and the locale:                 (3)
    MessageFormat mf = new MessageFormat(bundle.getString("pattern"),
                                         requestedLocale);

    // Argument values:                                                      (4)
    String itemName = bundle.getString("item.name");
    double itemPrice = Double.parseDouble(bundle.getString("item.price"));
    int numOfItems = 1234;
    Date timeOnly = ConvertToLegacyDate.ltToDate(LocalTime.of(14,30));
    Date dateOnly = ConvertToLegacyDate.ldToDate(LocalDate.of(2021,3,1));

    // Create argument array:                                                (5)
    Object[] messageArguments = {
        itemName,       // {0}
        itemPrice,      // {1,number,currency}
        numOfItems,     // {2,number,integer}
        timeOnly,       // {3,time,short}
        dateOnly,       // {4,date,long}
    };

    // Apply the formatter to the arguments:                                 (6)
    String result = mf.format(messageArguments);
    System.out.println(result);
  }

  public static void main(String[] args) {
    displayStockInfo(Locale.US);
    System.out.println();
    displayStockInfo(new Locale("es", "ES"));
    System.out.println();
  }
}
```

```
Requested Locale: en_US
Stock date: 2:30 PM, March 1, 2021
Item name: Frozen pizza
Item price: $9.99
Number of items: 1,234

Requested Locale: es_ES
Fecha de stock: 14:30, 1 de marzo de 2021
```

```
Nombre del árticulo: Pizza congelada
Precio del articulo: 8,99 ?
Número de artículos: 1.234
```

```
MessageFormat mf = new MessageFormat(previousPattern);
mf.setLocale(newLocale);
mf.applyPattern(newPattern);
String output = mf.format(newMessageArguments);
```

```
"There {0,choice,0#are no bananas|1#is only one banana|\
                                 2<are {1,number,integer} bananas}."
```

```
"There are {1,number,integer} bananas."
```

```
# File: ChoiceBundle.properties
choice.pattern = There {0,choice,0#are no bananas|1#is only one banana|\
                                 2<are {1,number,integer} bananas}.
...
```

```
import java.text.*;
import java.util.*;

public class ChoicePatternUsage {

  static void displayMessages(Locale requestedLocale) {
    System.out.println("Requested Locale: " + requestedLocale);

    // Fetch the resource bundle:                                          (1)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // choice.pattern = There {0,choice,0#are no bananas|1#is only one banana
    //                                 |2#are {1,number,integer} bananas}.
    // Create formatter for specified choice pattern and locale:           (2)
    MessageFormat mf =
        new MessageFormat(bundle.getString("choice.pattern"), requestedLocale);

    // Create the message argument arrays:                                 (3)
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:                                  (4)
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);        // (5)
      System.out.printf("Arguments:%-10sResult: %s%n",
                Arrays.toString(messageArguments[choiceNumber]),result);
    }
  }

  public static void main(String[] args) {
    displayMessages(new Locale("en", "US"));
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

```
Requested Locale: en_US
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested Locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

```
ChoiceFormat(double[] limits, String[] formats)
```

`ChoiceFormat(String newPattern)`

```
# File: ChoiceBundle.properties
...
pattern = There {0}.
none = are no bananas
singular = is only one banana
plural = are {1,number,integer} bananas
```

```
import java.text.*;
import java.util.*;

public class ChoiceFormatUsage {

  static void displayMessages(Locale requestedLocale)  {
    System.out.println("Requested locale: " + requestedLocale);

    // Fetch the resource bundle:                                            (1)
    ResourceBundle bundle =
        ResourceBundle.getBundle("resources.ChoiceBundle", requestedLocale);

    // Create formatter for specified pattern and locale:                    (2)
    MessageFormat mf = new MessageFormat(
        bundle.getString("pattern"),       // pattern = There {0}.
        requestedLocale
        );
    // Create the limits and the formats arrays:                               (3)
    double[] limits = {0,1,2};                                              // (3a)
    String [] grammarFormats = {                                            // (3b)
        bundle.getString("none"),       // none = are no bananas
        bundle.getString("singular"),   // singular = is only one banana
        bundle.getString("plural")      // plural = are {1,number,integer} bananas
    };

    // Create the choice format:                                               (4)
    ChoiceFormat choiceForm = new ChoiceFormat(limits, grammarFormats);

    // Create the formats:                                                     (5)
    Format[] formats = {choiceForm};

    // Set the formats in the formatter:                                       (6)
    mf.setFormats(formats);                                                 // (6a)
//  messageForm.setFormatsByArgumentIndex(formats);                         // (6b)
//  mf.setFormat(0, choiceForm);                                            // (6c)

    // Create the arguments arrays:                                            (7)
    Object[][] messageArguments = { {0.5}, {1.5}, {2.5, 2} };

    // Test the formatter with arguments:                                      (8)
    for (int choiceNumber = 0;
         choiceNumber < messageArguments.length; choiceNumber++) {
      String result = mf.format(messageArguments[choiceNumber]);            // (9)
      System.out.printf("Arguments:%-10sResult: %s%n",
          Arrays.toString(messageArguments[choiceNumber]),result);    }
  }

  public static void main(String[] args)   {
    displayMessages(new Locale("en", "US"));
    System.out.println();
    displayMessages(new Locale("es", "ES"));
  }
}
```

```
Requested Locale: en_US
Arguments:[0.5]     Result: There are no bananas.
Arguments:[1.5]     Result: There is only one banana.
Arguments:[2.5, 2]  Result: There are 2 bananas.

Requested locale: es_ES
Arguments:[0.5]     Result: Ahí no hay plátanos.
Arguments:[1.5]     Result: Ahí es solo un plátano.
Arguments:[2.5, 2]  Result: Ahí son 2 plátanos.
```

```
Object[] parse(String source) throws ParseException
Object[] parse(String source, ParsePosition position)
```

```
String pattern = "foo {0,number,currency} bar";                       // (1)
MessageFormat mfp = new MessageFormat(pattern, Locale.US);
Object[] arguments = new Object[] {10.99};            // [10.99]
String formattedResult = mfp.format(arguments);       // "foo $10.99 bar"
Object[] parsedResult = mfp.parse(formattedResult);   // [10.99]
System.out.println((double)arguments[0] == (double)parsedResult[0]); // true
```

```
String patternA = "{0}";                                        // (2)
MessageFormat mfpA = new MessageFormat(patternA, Locale.US);
Object[] parsedResultA = mfpA.parse("One Two Three");           // (3)
System.out.println(parsedResultA[0] instanceof String);   // true
System.out.println(Arrays.toString(parsedResultA));       // (4) [One Two Three]
```

```
String patternB = "One {0} Three";                                   // (5)
MessageFormat mfpB = new MessageFormat(patternB, Locale.US);
Object[] parsedResultB = mfpB.parse("One 2 Three");                  // (6)
System.out.println(parsedResultB[0].equals("2"));                    // true
System.out.println(Arrays.toString(parsedResultB));                  // [2]
```

```
Object[] parsedResultC = mfpB.parse("One 2 Three whatever");          // (7)
System.out.println(Arrays.toString(parsedResultC));                   // [2]
```

```
String patternD = "foo {0,number,integer} {1,number,currency} bar";  // (8)
MessageFormat mfpD = new MessageFormat(patternD, Locale.US);
Object[] parsedResultD = mfpD.parse("foo 2021 $64.99 bar");          // (9)
System.out.println(Arrays.toString(parsedResultD));         // [2021, 64.99]
```

```
Locale locale = new Locale("no", "NO");
String pattern1 = "foo {0,number,currency} bar";
MessageFormat mfp1 = new MessageFormat(pattern1, locale);
// String parseSource = "foo kr 10,99 bar";               // (10a) ParseException
String parseSource = "foo kr\u00a010,99 bar";             // (10b) nbsp. OK.
Object[] parsedResults = mfp1.parse(parseSource);
if (parsedResults[0] instanceof Double dValue) {
  System.out.println(dValue);                             // 10.99
}
```

```
format(String format, Object... args)
format(Locale loc, String format, Object... args)
```

```
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Locale;

public class DecimalNumberPatternsRQ {
  public static void main(String[] args) {
    // (1) INSERT CODE HERE
    double value = 0.456;
    DecimalFormat df = (DecimalFormat) NumberFormat.getNumberInstance(Locale.US);
    df.applyPattern(pattern);
    String output = df.format(value);
    System.out.printf("|%s|", output);
  }
}
```

```
import java.math.*;
import java.text.*;
import java.util.*;

public class RQ7 {
  public static void main(String[] args) {
    NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);
    nf.setMaximumFractionDigits(2);
    nf.setRoundingMode(RoundingMode.HALF_DOWN);
    double value = 9876.54321;
    String s1 = nf.format(value);
    nf.setRoundingMode(RoundingMode.HALF_UP);
    String s2 = nf.format(value);
    System.out.println(s1 + " " + s2);
  }
}
```

```
import java.math.*;
import java.text.*;
import java.util.*;

public class RQ7Alt {
  public static void main(String[] args) {
    BigDecimal value = new BigDecimal("9876.545");
    NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US);
    nf.setMaximumFractionDigits(2);
    nf.setRoundingMode(RoundingMode.HALF_DOWN);
    String s1 = nf.format(value);
    System.out.println();
    nf.setRoundingMode(RoundingMode.HALF_UP);
    String s2 = nf.format(value);
    System.out.println(s1+" "+s2);
  }
}
```

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class RQ8 {
  public static void main(String[] args) {
    LocalDate foolsDay = LocalDate.of(2021, Month.APRIL, 1);
    DateTimeFormatter df = DateTimeFormatter
                               .ofLocalizedDateTime(FormatStyle.SHORT)
                               .localizedBy(Locale.UK);
    System.out.print(df.format(foolsDay));
  }
}
```

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class RQ9 {
  public static void main(String[] args) {
    LocalDateTime foolsDay = LocalDateTime.of(2021, Month.APRIL, 1, 14, 30, 0);
    DateTimeFormatter df = DateTimeFormatter.ofPattern("day")
                                            .localizedBy(Locale.UK);
    System.out.print(df.format(foolsDay));
  }
}
```

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class RQ10 {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(2021, Month.APRIL, 1);
    Locale.setDefault(Locale.UK);
    DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
    String s1 = df.format(d);
    Locale.setDefault(Locale.FRANCE);
    String s2 = df.format(d);
    df.localizedBy(Locale.US);
    String s3 = df.format(d);
    if("Apr 1, 2021".equals(s1)) {
      System.out.print("UK ");
    }
    if("1 avr. 2021".equals(s2)) {
      System.out.print("FR ");
    }
    if("1 Apr 2021".equals(s3)) {
      System.out.println("US");
    }
  }
}
```

```
import java.text.*;
import java.util.*;

public class RQ11 {
  public static void main(String[] args) {
    double x = 0.987654321;
    NumberFormat nf = NumberFormat.getPercentInstance(Locale.US);
```

```
      System.out.println(nf.format(x));
    }
  }
```

```
import java.text.*;
import java.time.*;
import java.time.format.*;

public class RQ12 {
  public static void main(String[] args) {
    ZoneId london = ZoneId.of("Europe/London");
    ZoneId paris = ZoneId.of("Europe/Paris");
    LocalDateTime date1 = LocalDateTime.parse("2021-01-01T01:01:01",
                                             DateTimeFormatter.ISO_DATE_TIME);
    ZonedDateTime date2 = date1.atZone(london);
    DateTimeFormatter df1 = DateTimeFormatter.ofPattern("hz")
                                            .withZone(london);
    DateTimeFormatter df2 = DateTimeFormatter.ofPattern("hz")
                                            .withZone(paris);
    String result = MessageFormat.format("{0} {1} {2} {3}",
                                        df1.format(date1), df1.format(date2),
                                        df2.format(date1), df2.format(date2));
    System.out.println(result);
  }
}
```

```
import java.text.MessageFormat;

public class RQ13 {
  public static void main(String[] args) {
    String a = "A", b = "B";
    String result = MessageFormat.format("{0}-'{1}'-{3}-{0}-{1}-'{2}'", a, b);
    System.out.println(result);
  }
}
```

```
import java.text.*;

public class RQ14 {
  public static void main(String[] args) {
    double[] limits = {0,1,2,3,4};
    String[] formats = {"zero","{1}st","{1}nd","{1}rd","{1}th"};
    ChoiceFormat cf = new ChoiceFormat(limits, formats);
    MessageFormat mf = new MessageFormat("{0}");
    mf.setFormat(0, cf);
    Object[] values = {4,5};
    System.out.println(mf.format(values));
  }
}
```

```
import java.text.*;

public class RQ15 {
  public static void main(String[] args) {
    double[] limits = {0,-1,1};
    String[] formats = {"zero","negative","positive"};
    ChoiceFormat cf = new ChoiceFormat(limits, formats);
    MessageFormat mf = new MessageFormat("{0}");
    mf.setFormat(0, cf);
    Object[] values = {0.9};
    System.out.println(mf.format(values));
  }
}
```

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class RQ16 {
  public static void main(String[] args) {
    Locale.setDefault(Locale.US);
    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
    LocalDate d = LocalDate.parse("Apr 1, 2021", dtf);  // (1)
    d = LocalDate.parse("2021-04-01");                  // (2)
    String s = d.format(dft);                           // (3)
    System.out.println(s);
  }
}
```

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class RQ17 {
  public static void main(String[] args) {
    Locale.setDefault(Locale.US);
    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
    LocalDate d = LocalDate.parse("2021-04-01");    // (1)
    d = LocalDate.parse("4/1/21", dtf);              // (2)
    System.out.println(d);
  }
}
```

```
module moduleName {
  // Body:
  requires moduleName_1;          // A requires directive.
  ...
  requires moduleName_n;
  exports packageName_1;          // An exports directive.
  ...
  exports packageName_m;
}
```

```
module com.abc.client {
  requires com.abc.seller;
}
```

```
module com.abc.seller {
  exports com.abc.seller;
  exports com.abc.factory;
}
```

```
my.big.fat.module, com.passion.controller, org.geek.gui, view
```

`net.soda.7up, org:factory, com-factory-gizmo`

```
    requires seller;                                     // (1)
```

```
  exports com.abc.seller;                         // (2)
```

```
public class GizmoSeller { /*...*/ }          // (3)
```

```
module moduleX {
  exports packageOne to moduleA, moduleB;   // (1) Qualified export.
  exports packageTwo;                        // (2) Unqualified export.
}
```

```
// File path: src/model/com/passion/model/AdviceModel.java                  (1)
package com.passion.model;

public class AdviceModel {

  private String currentAdvice;

  public AdviceModel() { this.setCurrentAdvice(0); }

  public void setCurrentAdvice(int i) {
    String advice;
    switch(i) {
      case 1 : advice = "See no evil."; break;
      case 2 : advice = "Speak no evil."; break;
      case 3 : advice = "Hear no evil."; break;
      default: advice = "No advice.";
    }
    currentAdvice = advice;
  }

  public String getCurrentAdvice() { return currentAdvice; }
}
```

```
// File path: src/view/com/passion/view/AdviceView.java                    (2)
package com.passion.view;
import com.passion.model.AdviceModel;              // From model module.

public class AdviceView {

  private AdviceModel model;

  public AdviceView(AdviceModel model) { this.model = model; }

  public void updateAdviceDisplay(){
    System.out.println(model.getCurrentAdvice());
  }
}
```

```
// File path: src/controller/com/passion/controller/AdviceController.java   (3)
package com.passion.controller;
import com.passion.model.AdviceModel;                   // From model module.
import com.passion.view.AdviceView;                     // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);                       // Inject the model.
  }

  public void showAdvice(int adviceNumber) {
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

```
// File path: src/main/com/passion/main/Main.java                          (4)
package com.passion.main;
import com.passion.controller.AdviceController;    // From controller module.

public class Main {
  public static void main(String... args) {
    AdviceController controller = new AdviceController();
    controller.showAdvice(1);
    controller.showAdvice(2);
    controller.showAdvice(3);
    controller.showAdvice(0);
  }
}
```

```
>javac --module-path mods \
       -d mods/model \
       src/model/module-info.java \
       src/model/com/passion/model/AdviceModel.java
```

```
>javac --module-source-path src -d mods --module main
```

```
>java --module-path mods --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

```
added manifest
added module-info: module-info.class
adding: com/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/model/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/model/AdviceModel.class(in = 641) (mods= 415)(deflated 35%)
```

```
>jar --verbose \
     --create \
     --file mlib/main.jar \
     --main-class com.passion.main.Main \
     -C mods/main .
```

```
>jar -vcfe mlib/main.jar com.passion.main.Main -C mods/main .
```

```
>java --module-path mlib --module main
```

```
>java --module-path mlib --module main/com.passion.main.Main
```

```
module org.liberal {
  opens org.liberal.sesame;     // An open package
}
```

```
// Filepath: src/controller/module-info.java
module controller {                                  // (1)
  requires model;
  requires view;
  opens com.passion.controller;                      // Open package
}
```

```
// Filepath: src/main/module-info.java
module main {}                                             // (2)
```

```
// File path: src/main/com/passion/main/Main.java
package com.passion.main;
import java.lang.reflect.*;                                 // Types for reflection

public class Main {
  public static void main(String... args) {                           // (1)
    try {
      // Get the runtime object representing the class.
      Class<?> cObj
          = Class.forName("com.passion.controller.AdviceController");      // (2)

      // Get the no-argument constructor of the class.
      Constructor<?> constructor = cObj.getDeclaredConstructor();          // (3)

      // Create an instance of the class.
      Object instance = constructor.newInstance();                         // (4)

      // Get all declared methods, including those that are non-public.
      Method[] methods = cObj.getDeclaredMethods();                        // (5)
      // Class has only one method.
      Method method = methods[0];                                          // (6)
      // Check if it is the right method.
      if (!method.getName().equals("showAdvice")) {                        // (7)
        System.out.printf("Method showAdvice() not found in %s.%n",
                          cObj.getName());
        return;
      }

      // Disable access control checks on the method as it is a private method.
      method.setAccessible(true);                                          // (8)

      // Invoke the method on the instance, passing any arguments.
      method.invoke(instance, 1);                                          // (9)
      method.invoke(instance, 2);                                          // (10)
      method.invoke(instance, 3);                                          // (11)
      method.invoke(instance, 0);                                          // (12)

    } catch (ClassNotFoundException | NoSuchMethodException |
             InstantiationException | IllegalAccessException |
             IllegalArgumentException | InvocationTargetException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
// File path: src/controller/com/passion/controller/AdviceController.java
package com.passion.controller;

import com.passion.model.AdviceModel;                // From model module.
import com.passion.view.AdviceView;                  // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);
  }

  private void showAdvice(int adviceNumber) {        // (13)
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

```
>javac -d mods --module-source-path src --module model,view,controller,main
```

```
>java --module-path mods --module main/com.passion.main.Main
java.lang.ClassNotFoundException: com.passion.controller.AdviceController
...
```

```
>java --add-modules controller --module-path mods \
      --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

```
module org.liberal {
  opens org.liberal.sesame to com.aladdin, forty.thieves; // (1) Qualified opens
  opens org.liberal.pandorasbox;                          // (2) Unqualified opens
}
```

```
module controller {
  requires model;
  requires view;
  opens com.passion.controller to main;   // Package is only open to main module.
}
```

```
open module module_name {
  // Any number of module directives, except the opens directive.
}
```

```
open module controller {                // An open module
  requires model;
  requires view;
}
```

```
// File: ./src/serviceinterface/org/advice/si/IAdvice.java
package org.advice.si;
import java.util.Locale;

// Service interface
public interface IAdvice {
  String getContent();                      // (1) Returns the content of the advice.
  Locale getLocale();                       // (2) Returns the associated locale.
}
```

```
// File: ./src/serviceinterface/module-info.java
module serviceinterface {
   exports org.advice.si;
}
```

```
// File:./src/serviceprovider/org/advice/serviceprovider/CannedAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class CannedAdvice implements IAdvice {
  public String getContent() {                          // (1)
    return "Keep calm and service on!";
  }
  public Locale getLocale() { return Locale.UK; }      // (2)
}
```

```
// File:./src/serviceprovider/org/advice/serviceprovider/VikingAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class VikingAdvice implements IAdvice {
  public String getContent() {                                       // (3)
    return "Gi aldri opp!";    // Never give up!
  }
  public Locale getLocale() { return new Locale("no", "Norway"); }  // (4)
}
```

```
// File: ./src/serviceprovider/module-info.java
module serviceprovider {
  requires serviceinterface;
  provides org.advice.si.IAdvice with              // (1)
    org.advice.serviceprovider.CannedAdvice,
    org.advice.serviceprovider.VikingAdvice;
}
```

```
ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream();
```

```
static <S> ServiceLoader<S> load(Class<S> service)
```

```
Stream<ServiceLoader.Provider<S>> stream()
```

```
// File:./src/servicelocator/org/advice/servicelocator/AdviceLocator.java
package org.advice.servicelocator;

import java.util.*;
import java.util.stream.*;
import org.advice.si.*;

public class AdviceLocator {

  /** Get advice for a particular locale. */
  public static Optional<IAdvice> getAdvice(Locale desiredLocale) {     // (1)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);  // (2)
    Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream(); // (3)
    Optional<IAdvice> optAdvice = spStream                              // (4)
      .map(ServiceLoader.Provider::get)                  // (5) Stream<IAdvice>
      .filter(a -> desiredLocale.equals(a.getLocale()))   // (6)
      .findFirst();                                       // (7)
    return optAdvice;
  }

  /** Get all advice implemented by all providers. */
  public static List<IAdvice> getAllAdvice() {            // (8)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
    List<IAdvice> allAdvice = loader
      .stream()
      .map(p -> p.get())                       // (9) map(ServiceLoader.Provider::get)
      .collect(Collectors.toList());           // (10) List<IAdvice>
    return allAdvice;
  }
}
```

```
// File: ./src/servicelocator/module-info.java
module servicelocator {
  requires serviceinterface;          // (1)
  exports org.advice.servicelocator;  // (2)
  uses org.advice.si.IAdvice;         // (3) In module serviceinterface.
}
```

```
// File: ./src/serviceconsumer/org/advice/serviceconsumer/AdviceConsumer.java
package org.advice.serviceconsumer;

import java.util.*;
import org.advice.servicelocator.*;
import org.advice.si.*;

public class AdviceConsumer {
  public static void main(String[] args) {

    // Get advice for the UK locale.                                    // (1)
    Optional<IAdvice> optAdvice = AdviceLocator.getAdvice(Locale.UK);   // (2)
    String adviceStr = optAdvice.map(IAdvice::getContent)               // (3)
                                .orElse("Sorry. No Advice!");           // (4)
    System.out.println("Advice for UK locale: " + adviceStr);

    // Get all implemented advice.                                      // (5)
    System.out.println("Printing all advice:");
    List<IAdvice> allAdvice = AdviceLocator.getAllAdvice();             // (6)
    allAdvice.stream()                                                  // (7)
      .forEach(a -> System.out.println(a.getContent()));
  }
}
```

```
// File: ./src/serviceconsumer/module-info.java
module serviceconsumer {
  requires serviceinterface;     // (1)
  requires servicelocator;       // (2)
}
```

```
>javac -d mods \
   --module-source-path src \
   --module serviceprovider,serviceconsumer,servicelocator,serviceinterface
```

```
>java --module-path mods \
      --module serviceconsumer/org.advice.serviceconsumer.AdviceConsumer
Advice for UK locale: Keep calm and service on!
Printing all advice:
Keep calm and service on!
Gi aldri opp!
```

```
jlink --module-path mlib --add-modules model,view,controller,main,java.base \
      --output advice
```

```
>advice/bin/java --list-modules
controller
java.base@17.0.2
main
model
view
```

```
>advice/bin/java --module main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

```
>jar --list --file mlib/main.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/
com/passion/
com/passion/main/
com/passion/main/Main.class
```

```
>jar --extract --file ../mlib/main.jar
>tree -F --dirsfirst --noreport main-extracted
main-extracted
├── META-INF/
│   └── MANIFEST.MF
├── com/
│   └── passion/
│       └── main/
│           └── Main.class
└── module-info.class
```

```
>more main-extracted/META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 17.0.2 (Oracle Corporation)
Main-Class: com.passion.main.Main
```

```
>java --module-path mlib --list-modules
java.base@17.0.2
...
jdk.javadoc@17.0.2
...
jdk.jdeps@17.0.2
...
controller file:.../adviceApp/mlib/controller.jar
main file:.../adviceApp/mlib/main.jar
model file:.../adviceApp/mlib/model.jar
view file:.../adviceApp/mlib/view.jar
```

```
>java --module-path mlib --describe-module main
main file:.../adviceApp/mlib/main.jar
requires java.base mandated
requires controller
contains com.passion.main
```

```
>jar --file mlib/main.jar --describe-module
main jar:file:.../adviceApp/mlib/main.jar/!module-info.class
requires controller
requires java.base mandated
contains com.passion.main
main-class com.passion.main.Main
```

```
>java --describe-module java.base
java.base@17.0.2
exports java.io
exports java.lang
...
uses java.util.spi.CurrencyNameProvider
uses java.util.spi.TimeZoneNameProvider
...
qualified exports jdk.internal to jdk.jfr
qualified exports sun.net.www to java.desktop java.net.http jdk.jartool
...
contains com.sun.crypto.provider
contains com.sun.java.util.jar.pack
...
```

```
>jdeps mods/model
model
[file:.../adviceApp/mods/model/]
   requires mandated java.base (@17.0.2)
model -> java.base
   com.passion.model           -> java.lang           java.base
```

```
>jdeps mlib/main.jar
Exception in thread "main" java.lang.module.FindException: Module controller
not found, required by main
       at java.base/...
```

```
>jdeps --module-path mlib --module main
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
   requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main     -> com.passion.controller     controller
   com.passion.main     -> java.lang                  java.base
```

```
>jdeps --module-path mlib --module main --recursive
controller
...
```

```
main

...
model

...
view

...
```

```
>jdeps --module-path mlib --module main -summary
main -> controller
main -> java.base
```

```
>jdeps --module-path mlib --module main -summary --recursive              (
controller -> java.base
controller -> model
controller -> view
main -> controller
main -> java.base
model -> java.base
view -> java.base
view -> model
```

```
>jdeps -dotoutput . -summary --recursive  mlib/*
>dot -Tpdf summary.dot >summary.pdf
```

```
>jdeps --module-path mlib --module main -verbose
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
   requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main.Main  -> com.passion.controller.AdviceController  controller
   com.passion.main.Main  -> java.lang.Object                         java.base
   com.passion.main.Main  -> java.lang.String                         java.base
```

```
>jdeps --module-path mlib --module main -verbose --recursive
controller
...
main
...
model
...
view
...
```

```
javac --module-path modulepath -d directory sourceAndModuleInfoFiles
javac  -p           modulepath -d directory sourceAndModuleInfoFiles
javac --module-source-path modulepath -d directory \
      --module moduleName1,...
javac --module-source-path modulepath -d directory \
       -m moduleName1,...
```

```
java --module-path modulepath --module moduleName/qualifiedClassName
java  -p          modulepath  -m      moduleName/qualifiedClassName
java --module-path modulepath  -module moduleName
java  -p          modulepath  -m      moduleName
```

```
jar --verbose --create --file jarfile -C directory files
jar -vcf jarfile -C directory .
jar -vcf jarfile --main-class qualifiedMainClassName -C directory .
jar -vcfe jarfile qualifiedMainClassName -C directory .

jar --list --file jarfile
jar -tf jarfile
```

```
java --list-modules
java --module-path modulepath --list-modules
java  -p          modulepath --list-modules
java --module-path modulepath --describe-module moduleName
java  -p            modulepath  -d                moduleName

jar --file jarFile --describe-module
jar  -f    jarFile  -d
```

```
jdeps --module-path modulepath -m moduleName
jdeps --module-path modulepath -m moduleName --recursive
jdeps --module-path modulepath -m moduleName -R
jdeps --module-path modulepath -m moduleName -summary
jdeps --module-path modulepath -m moduleName -summary --recursive
jdeps --module-path modulepath -m moduleName -s -R
jdeps --module-path modulepath -m moduleName -verbose
jdeps --module-path modulepath -m moduleName -verbose --recursive
jdeps --module-path modulepath -m moduleName -v -R
```

```
java --module-path path --module module[/mainclass]
```

```
jar --create --file jarfile --main-class qualifiedMainClassName -C DIR .
```

```
--main-class qualifiedMainClassName
--main-class=qualifiedMainClassName
 -e qualifiedMainClassName
 -e=qualifiedMainClassName
```

```
jlink --module-path path --add-modules module_names --output output_dir
```

```
module store {
  requires transitive product;
  exports store.frontend to ui;
  exports store.backend;
}
```

```
module product {
  exports product.data;
}

module ui {
  requires store;
}

module customer {
  requires ui;
  exports customer.data;
}
```

```
module zoo {
  requires animals;
}

module animals {
  exports animals.primates;
}
```

```
package animals.primates;
public class Chimpanzee extends Ape { }
```

```
module store {
  requires product;
  exports store.frontend;
}

module marketing {
  requires store;
  opens marketing.offers;
}

module product {
  exports product.data;
  exports product.pricing to marketing;
}
```

```
module music {
  requires artist.recording;
  requires production.mix;
}
module production {
  opens production.mix;
  exports production.recording;
}
```

```
module artist {
  exports artist.recording;
}
```

```
module music {
  requires production;
}
module production {
  opens production.mix;
  exports recording;
}
module artist {
  requires transitive production;
  exports artist.recording;
}
```

```
module music {
  requires production.recording;
}
module production {
  opens music;
  exports production.mix;
}
module artist {
  requires transitive production.recording;
  exports artist.recording;
}
```

```
module music {
  exports music.sound;
}

module brass {
  requires music;
  provides music.sound.Instrument with brass.sound.Trumpet;
}
```

```
module player {
  requires brass;
  uses music.sound.Instrument;
}
```

```
module player {
  requires music;
  uses music.sound.Instrument;
}
```

```
module music {
  requires artist;
  requires instrument;
  exports preferences.style to catalog;
}

module instrument {
  requires transitive music;
  opens instrument.data;
}

module artist {
  exports preferences.style;
}

module catalog {
}
```

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

```
long transferTo(OutputStream out) throws IOException
```

```
void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
```

`void close() throws IOException`	Both `InputStream` and `OutputStream`
`void flush() throws IOException`	Only for `OutputStream`

```
FileInputStream(String name) throws FileNotFoundException
```

```
FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String name, boolean append) throws FileNotFoundException
```

```
/* Copy a file using a byte buffer.
   Command syntax: java CopyFile <from_file> <to_file> */
import java.io.*;

class CopyFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileInputStream fromFile = new FileInputStream(args[0]);       // (1)
        FileOutputStream toFile = new FileOutputStream(args[1]))  {    // (2)

      // Copy bytes using buffer:                                      // (3a)
      byte[] buffer = new byte[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer bytes:
//    fromFile.transferTo(toFile);                                     // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyFile <from_file> <to_file>");
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

```
DataInputStream(InputStream in)
DataOutputStream(OutputStream out)
```

```
FileOutputStream outputFile = new FileOutputStream("primitives.data");
```

```
DataOutputStream outputStream = new DataOutputStream(outputFile);
```

```
FileInputStream inputFile = new FileInputStream("primitives.data");
```

```
DataInputStream inputStream = new DataInputStream(inputFile);
```

```
import java.io.*;

public class BinaryValuesIO {
  public static void main(String[] args) throws IOException {

    // Write binary values to a file:
    try(                                                                  // (1)
        // Create a FileOutputStream.                                        (2)
        FileOutputStream outputFile = new FileOutputStream("primitives.data");
        // Create a DataOutputStream which is chained to the FileOutputStream.(3)
        DataOutputStream outputStream = new DataOutputStream(outputFile)) {

      // Write Java primitive values in binary representation:               (4)
      outputStream.writeBoolean(true);
      outputStream.writeChar('A');               // int written as Unicode char
      outputStream.writeByte(Byte.MAX_VALUE);    // int written as 8-bits byte
      outputStream.writeShort(Short.MIN_VALUE);  // int written as 16-bits short
      outputStream.writeInt(Integer.MAX_VALUE);
      outputStream.writeLong(Long.MIN_VALUE);
      outputStream.writeFloat(Float.MAX_VALUE);
      outputStream.writeDouble(Math.PI);
    }

    // Read binary values from a file:
    try (                                                                 // (1)
        // Create a FileInputStream.                                         (2)
        FileInputStream inputFile = new FileInputStream("primitives.data");
        // Create a DataInputStream which is chained to the FileInputStream.  (3)
        DataInputStream inputStream = new DataInputStream(inputFile)) {

      // Read the binary representation of Java primitive values
      // in the same order they were written out:                          (4)
      System.out.println(inputStream.readBoolean());
      System.out.println(inputStream.readChar());
      System.out.println(inputStream.readByte());
      System.out.println(inputStream.readShort());
      System.out.println(inputStream.readInt());
      System.out.println(inputStream.readLong());
      System.out.println(inputStream.readFloat());
      System.out.println(inputStream.readDouble());

      // Check for end of stream:
      int value = inputStream.readByte();
      System.out.println("More input: " + value);
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found.");
    } catch (EOFException eof) {
      System.out.println("End of input stream.");
    }
  }
}
```

```
static Charset forName(String charsetName)
static Charset defaultCharset()
```

```
int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws IOException
```

```
long skip(long n) throws IOException
```

```
void close() throws IOException
```

```
boolean ready() throws IOException
```

```
long transferTo(Writer out) throws IOException
```

```
void write(int c) throws IOException
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int length) throws IOException
void write(String str, int off, int length) throws IOException
void close() throws IOException
void flush() throws IOException
```

```
/* Copy a file using a character buffer.
   Command syntax: java CopyCharacterFile <from_file> <to_file> */
import java.io.*;

class CopyCharacterFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileReader fromFile = new FileReader(args[0]);                  // (1)
        FileWriter toFile = new FileWriter(args[1]))  {                 // (2)

      // Copy characters using buffer:                                  // (3a)
      char[] buffer = new char[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer characters:
//    fromFile.transferTo(toFile);                                      // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyCharacterFile <from_file> <to_file>");
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName) throws FileNotFoundException
PrintWriter(String fileName, Charset charset)
            throws FileNotFoundException
PrintWriter(String fileName, String charsetName)
            throws FileNotFoundException, UnsupportedEncodingException
```

```
PrintWriter format(String format, Object... args)
PrintWriter format(Locale loc, String format, Object... args)

PrintWriter printf(String format, Object... args)
PrintWriter printf(Locale loc, String format, Object... args)
```

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
```

```
OutputStreamWriter outputStream = new OutputStreamWriter(outputFile);
```

```
PrintWriter printWriter1 = new PrintWriter(outputStream, true);
```

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
```

```
PrintWriter printWriter2 = new PrintWriter(outputFile, true);
```

```
FileWriter fileWriter = new FileWriter("info.txt");
```

```
PrintWriter printWriter3 = new PrintWriter(fileWriter, true);
PrintWriter printWriter4 = new PrintWriter("info.txt");
```

```
Charset utf8 = Charset.forName("UTF-8");
FileWriter fileWriter = new FileWriter("info.txt", utf8);
PrintWriter printWriter5 = new PrintWriter(fileWriter, true);
```

```
Charset utf8 = Charset.forName("UTF-8");
PrintWriter printWriter6 = new PrintWriter("info.txt", utf8);
```

```
FileInputStream inputFile = new FileInputStream("info.txt");
```

```
InputStreamReader reader = new InputStreamReader(inputFile);
```

```
FileReader fileReader = new FileReader("info.txt");
```

```
Charset utf8 = Charset.forName("UTF-8");
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader = new InputStreamReader(inputFile, utf8);
```

```
Charset utf8 = Charset.forName("UTF-8");
FileReader reader = new FileReader("info.txt", utf8);
```

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

```
Charset utf8 = Charset.forName("UTF-8");
FileOutputStream   outputFile      = new FileOutputStream("info.txt");
OutputStreamWriter outputStream    = new OutputStreamWriter(outputFile, utf8);
BufferedWriter     bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter        printWriter1    = new PrintWriter(bufferedWriter1, true);
```

```
FileWriter     fileWriter      = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter    printWriter2    = new PrintWriter(bufferedWriter2, true);
```

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

```
String readLine() throws IOException
```

```
// Using the UTF-8 character encoding:
Charset utf8 = Charset.forName("UTF-8");
FileReader     fileReader      = new FileReader("lines.txt", utf8);
BufferedReader bufferedReader1 = new BufferedReader(reader)

// Use the default encoding:
FileReader     fileReader      = new FileReader("lines.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader);
```

```
import java.io.*;
import java.nio.charset.Charset;
import java.time.LocalDate;

public class CharEncodingDemo {
  public static void main(String[] args)
         throws FileNotFoundException, IOException, NumberFormatException {

    // UTF-8 character encoding.
    Charset utf8 = Charset.forName("UTF-8");                           // (1)
    try(// Create a BufferedWriter that uses UTF-8 character encoding      (2)
        FileWriter writer = new FileWriter("info.txt", utf8);
        BufferedWriter bufferedWriter1 = new BufferedWriter(writer);
        PrintWriter printWriter = new PrintWriter(bufferedWriter1, true);) {

      System.out.println("Writing using encoding: " + writer.getEncoding());
      // Print some values, one on each line.                            (3)
      printWriter.println(LocalDate.now());
      printWriter.println(Integer.MAX_VALUE);
      printWriter.println(Long.MIN_VALUE);
      printWriter.println(Math.PI);
    }

    try(// Create a BufferedReader that uses UTF-8 character encoding      (4)
        FileReader reader = new FileReader("info.txt", utf8);
        BufferedReader bufferedReader = new BufferedReader(reader);) {

      System.out.println("Reading using encoding: " + reader.getEncoding());
      // Read the character input and parse accordingly.                 (5)
      LocalDate ld = LocalDate.parse(bufferedReader.readLine());
      int iMax = Integer.parseInt(bufferedReader.readLine());
      long lMin = Long.parseLong(bufferedReader.readLine());
      double pi = Double.parseDouble(bufferedReader.readLine());

      // Write the values read on the terminal                           (6)
      System.out.println("Values read:");
      System.out.println(ld);
      System.out.println(iMax);
      System.out.println(lMin);
      System.out.println(pi);

      // Check for end of stream:                                        (7)
      String line = bufferedReader.readLine();
      if (line != null ) {
        System.out.println("More input: " + line);
      } else {
        System.out.println("End of input stream");
      }
    }
  }
}
```

```
// Obtaining the console:
Console console = System.console();
if (console == null) {
  System.err.println("No console available.");
  return;
}
// Continue ...
```

```
String username = console.readLine("Enter the username (%d chars): ", 4);
```

```
char[] password;
do {
    password = console.readPassword("Enter password (min. %d chars): ", 6);
} while (password.length < 6);
```

```
String readLine()
String readLine(String format, Object... args)
```

```
char[] readPassword()
char[] readPassword(String format, Object... args)
```

```
Console format(String format, Object... args)
Console printf(String format, Object... args)
```

```
import java.io.Console;
import java.io.IOException;
import java.util.Arrays;

/** Class to change the password of a user */
public class ChangePassword {
  public static void main (String[] args) throws IOException  {

    // Obtain the console:                                           (1)
    Console console = System.console();
    if (console == null) {
      System.err.println("No console available.");
      return;
    }

    // Changing the password:                                        (2)
    boolean noMatch = false;
    do {
      // Read the new password and its confirmation:
      char[] newPasswordSelected
          = console.readPassword("Enter your new password: ");
      char[] newPasswordConfirmed
          = console.readPassword("Confirm your new password: ");

      // Compare the supplied passwords:
      noMatch = newPasswordSelected.length == 0  ||
                newPasswordConfirmed.length == 0 ||
                !Arrays.equals(newPasswordSelected, newPasswordConfirmed);
      if (noMatch) {
        console.format("Passwords don't match. Please try again.%n");
      } else {
        // Necessary code to change the password.
        console.format("Password changed.");
      }
    } while (noMatch);
  }
}
```

```
public static boolean test(InputStream is) throws IOException {
  int value = is.read();
  return value >= 0;
}
```

```
import java.io.*;
public class MoreEndings {
  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("seq.txt");
        InputStreamReader isr = new InputStreamReader(fis);) {
      int i = isr.read();
      while (i != -1) {
        System.out.print((char)i + "|");
        i = isr.read();
      }
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found");
    } catch (EOFException eofe) {
      System.out.println("End of stream");
    } catch (IOException ioe) {
      System.out.println("Input error");
    }
  }
}
```

```
import java.io.*;

public class NoEndings {
  public static void main(String[] args) {
    try (FileReader fr = new FileReader("greetings.txt");
         BufferedReader br = new BufferedReader(fr)) {
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
    } catch (EOFException eofe) {
      System.out.println("End of stream");
    } catch (IOException ioe) {
      System.out.println("Input error");
    }
  }
}
```

```
ObjectOutputStream(OutputStream out) throws IOException
```

```
FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
```

```
final void writeObject(Object obj) throws IOException
```

```
ObjectInputStream(InputStream in) throws IOException
```

```
FileInputStream inputFile = new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream = new ObjectInputStream(inputFile);
```

```
final Object readObject() throws ClassNotFoundException, IOException
```

```
import java.io.Serializable;
import java.time.Year;
/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Serializable {
```

```
  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

```
//Reading and Writing Objects
import java.io.*;
import java.time.Year;
import java.util.Arrays;

public class ObjectSerializationDemo {
  void writeData() {                                    // (1)
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      String[] strArray = {"Seven", "Eight", "Six"};
      long num = 2014;
      int[] intArray = {1, 3, 1949};
      String commonStr = strArray[2];                  // "Six"
      CD oneCD = new CD("Jaav", "Java Jive", 8, Year.of(2017), CD.Genre.POP);
      outputStream.writeObject(strArray);
      outputStream.writeLong(num);
      outputStream.writeObject(intArray);
      outputStream.writeObject(commonStr);
      outputStream.writeObject(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (IOException e) {
      System.err.println("Write error: " + e);
    }
  }

  void readData() {                                     // (2)
    try (// Set up the input stream:
        FileInputStream inputFile = new FileInputStream("obj-storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read the data:
      String[] strArray = (String[]) inputStream.readObject();
      long num = inputStream.readLong();
      int[] intArray = (int[]) inputStream.readObject();
      String commonStr = (String) inputStream.readObject();
      CD oneCD = (CD) inputStream.readObject();

      // Write data to the standard output stream:
      System.out.println(Arrays.toString(strArray));
      System.out.println(num);
      System.out.println(Arrays.toString(intArray));
      System.out.println(commonStr);
      System.out.println(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (EOFException e) {
      System.err.println("End of stream: " + e);
    } catch (IOException e) {
      System.err.println("Read error: " + e);
    } catch (ClassNotFoundException e) {
      System.err.println("Class not found: " + e);
    }
  }

  public static void main(String[] args) {
    ObjectSerializationDemo demo = new ObjectSerializationDemo();
    demo.writeData();
    demo.readData();
  }
}
```

```
[Seven, Eight, Six]
2014
[1, 3, 1949]
Six
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

```
class Wheel implements Serializable {                               // (1a)
  private int wheelSize;
  ...
}
```

```
class Unicycle implements Serializable {                                    // (2)
  private Wheel wheel;                                                      // (3a)
  ...
}
```

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

```
import java.io.Serializable;

// public class Wheel implements Serializable {                    // (1a)
public class Wheel {                                               // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

```
import java.io.Serializable;

public class Unicycle implements Serializable {                        // (2)
  private Wheel wheel;                                                 // (3a)
//transient private Wheel wheel;                                       // (3b)

  public Unicycle (Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }
}
```

```
import java.io.*;

public class SerialClient {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {

    try (// Set up the output stream:                              // (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
```

```
      // Write the data:
      Wheel wheel = new Wheel(65);
      Unicycle uc = new Unicycle(wheel);
      System.out.println("Before writing: " + uc);
      outputStream.writeObject(uc);
    }

    try (// Set up the input streams:                               // (5)
        FileInputStream inputFile = new FileInputStream("storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read data.
      Unicycle uc = (Unicycle) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + uc);
    }
  }
}
```

```
class Wheel implements Serializable {                                // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                             // (2)
  transient private Wheel wheel;                                     // (3b)
  ...
}
```

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with null
```

```
class Wheel {                                                      // (1b)
  private int wheelSize;
  ...
}
```

```
class Unicycle implements Serializable {                                    // (2)
  private Wheel wheel;                                                      // (3a)
  ...
}
```

```
>java SerialClient
Before writing: Unicycle with wheel size: 65
Exception in thread "main" java.io.NotSerializableException: Wheel
  ...
  at SerialClient.main(SerialClient.java:20)
```

```
private void writeObject(ObjectOutputStream) throws IOException;
private void readObject(ObjectInputStream)
                    throws IOException, ClassNotFoundException;
```

```
oos.defaultWriteObject();               // Method in the ObjectOutputStream class
oos.writeInt(wheel.getWheelSize());     // Method in the ObjectOutputStream class
```

```
ois.defaultReadObject();                // Method in the ObjectInputStream class
int wheelSize = ois.readInt();          // Method in the ObjectInputStream class
this.wheel = new Wheel(wheelSize);
```

```
public class Wheel {                                                    // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  public int getWheelSize() { return wheelSize; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

```
import java.io.*;

public class Unicycle implements Serializable {                          // (2)
  transient private Wheel wheel;                                         // (3b)

  public Unicycle(Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }

  private void writeObject(ObjectOutputStream oos) {                     // (3c)
    try {
      oos.defaultWriteObject();
      oos.writeInt(wheel.getWheelSize());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void readObject(ObjectInputStream ois) {                       // (3d)
    try {
      ois.defaultReadObject();
      int wheelSize = ois.readInt();
      this.wheel = new Wheel(wheelSize);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

```
public class SerialClient { // Same as in Example 20.7 }
```

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

```
Student student = new Student("Pendu", 1007);
System.out.println("Before writing: " + student);
outputStream.writeObject(student);
```

```
Student student = (Student) inputStream.readObject();
System.out.println("After reading: " + student);
```

```
Before writing: Student state(Pendu, 1007)
After reading: Student state(Pendu, 1007)
```

```
Before writing: Student state(Pendu, 1007)
No-argument constructor executed.
After reading: Student state(null, 1007)
```

```
public class Person  {                                            // (1b)
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

```
Before writing: Student state(Pendu, 1007)
Exception in thread "main" java.io.InvalidClassException:
        Student; no valid constructor
        ...
        at SerialInheritance.main(SerialInheritance.java:28)
```

```
import java.io.Serializable;

// A superclass
public class Person implements Serializable {                    // (1a)
//public class Person  {                                         // (1b)
  private String name;

  public Person() {                                              // (2)
    System.out.println("No-argument constructor executed.");
  }
  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

```
import java.io.Serializable;

public class Student extends Person {                              // (3a)
//public class Student extends Person implements Serializable {  // (3b)

  private long studNum;

  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }

  @Override
  public String toString() {
    return "Student state(" + getName() + ", " + studNum + ")";
  }
}
```

```
import java.io.*;

public class SerialInheritance {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {

    // Serialization:
    try (// Set up the output stream:                                      (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      Student student = new Student("Pendu", 1007);
      System.out.println("Before writing: " + student);
      outputStream.writeObject(student);
    }

    // Deserialization:
    try (// Set up the input stream:                                        (5)
        FileInputStream inputFile = new FileInputStream("storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read data.
      Student student = (Student) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + student);
    }
  }
}
```

```
// Original version of the Item class.
public class Item implements Serializable {                                 // (1)
  private double price;
//...
}
```

```
// New version of the Item class.
public class Item implements Serializable {                              // (2)
  private double price;
  private double weight;                                      // Additional field
//...
}
```

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = -4194294879924868414,
  local class serialVersionUID = -1186964199368835959
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

```
static final long serialVersionUID = 100L;            // Appropriate value.
```

```
// Original version of the Item class.
public class Item implements Serializable {    // (1)
  static final long serialVersionUID = 1000L;  // (1a)
  private double price;
//...
}

// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1000L;  // (2a) Same serialVersionUID
  private double price;
  private double weight;                       // Additional field
//...
}
```

```
Before writing: Price: 100.00
After reading: Price: 100.00, Weight: 0.00
```

```
// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1001L;  // (2b) Different serialVersionUID
  private double price;
  private double weight;
//...
}
```

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = 1000,
  local class serialVersionUID = 1001
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

```
import java.io.Serializable;

// Original version of the Item class.
public class Item implements Serializable {                                // (1)
//static final long serialVersionUID = 1000L;                              // (1a)

  private double price;

  public Item(double price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return String.format("Price: %.2f%n", this.price);
  }
}
```

```
import java.io.Serializable;

// New version of the Item class.
public class Item implements Serializable {                             // (2)
//static final long serialVersionUID = 1000L;                           // (2a)
//static final long serialVersionUID = 1001L;                           // (2b)

  private double price;
  private double weight;
  public Item(double price, double weight) {
    this.price = price;
    this.weight = weight;
  }
  @Override
  public String toString() {
    return String.format("Price: %.2f, Weight: %.2f", this.price, this.weight);
  }
}
```

```
// Serializer for objects of class Item.
import java.io.*;

public class Serializer {                                                  // (3)
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("item_storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Serialize an object of the original class:
      Item item = new Item(100.00);
      System.out.println("Before writing: " + item);
      outputStream.writeObject(item);
    }
  }
}
```

```
// Deserializer for objects of class Item.
import java.io.*;

public class DeSerializer{                                                  // (4)
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the input streams:
        FileInputStream inputFile = new FileInputStream("item_storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read a serialized object of the Item class.
      Item item = (Item) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + item);
    }
  }
}
```

```
public class Person  {
  protected String name;
  public Person() {this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
import java.io.Serializable;
public class Student extends Person implements Serializable {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
```

```
import java.io.*;
public class RQ800_10 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

```
import java.io.Serializable;
public class Person implements Serializable {
  protected String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
public class Student extends Person {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
```

```
import java.io.*;
public class RQ800_30 {
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

```
public class Person {
  protected transient String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
```

```
public class Student extends Person {
  protected long studNum;
  public Student() { }
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class GraduateStudent extends Student implements Serializable {
  private int year;
  public GraduateStudent(String name, long studNum, int year) {
    super(name, studNum);
    this.year = year;
  }
  public String toString() {
    return "(" + name + ", " + studNum + ", " + year  + ")";
  }

  private void readObject(ObjectInputStream ois)
      throws IOException, ClassNotFoundException {
    ois.defaultReadObject();
    this.name = "NewName";
    this.studNum = 200;
    this.year = 2;
  }
}
```

```
import java.io.*;
public class RQ800_70 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      GraduateStudent stud1 = new GraduateStudent("Aesop", 100, 1);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      GraduateStudent stud2 = (GraduateStudent) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

```
import java.io.Serializable;
public class Product implements Serializable {
  private String name;

  public Product(String name) {
    this.name = name;
    System.out.print("product ");
  }

  @Override
  public String toString() {
    return name;
  }
}
```

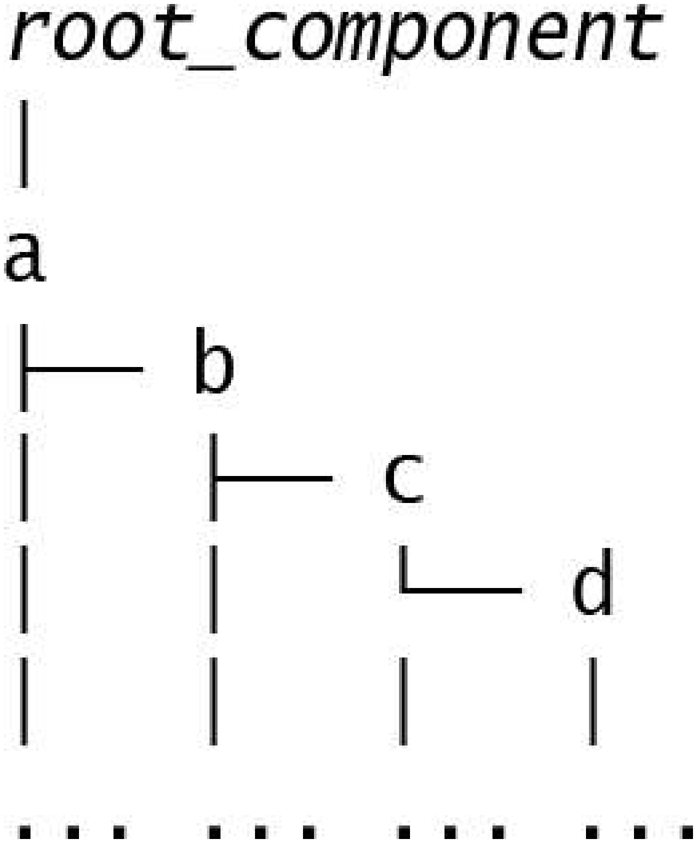
```
public class Food extends Product {
  private int calories;

  public Food(String name, int calories) {
    super(name);
    this.calories = calories;
    System.out.print("food ");
  }

  @Override
  public String toString() {
    return super.toString()+" "+calories;
  }
}
```

```
import java.io.*;
public class RQ20 {
  public static void main(String[] args) {
    Product p = new Food("cookie", 300);
    try(ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream("prod.dat"))) {
      out.writeObject(p);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try(ObjectInputStream in =
            new ObjectInputStream(new FileInputStream("prod.dat"))) {
      p = (Food)in.readObject();
    } catch (Exception ex) {
      System.out.println("error deserializing product");
    }
    System.out.println(p);
  }
}
```

```
import java.io.*;
public class RQ21 {
  public static void main(String[] args) {
    char[] buffer = new char[4];
    int count = 0;
    try(FileReader in = new FileReader("test1.txt");
        FileWriter out = new FileWriter("test2.txt")) {
      while((count = in.read(buffer)) != -1) {
        out.write(buffer);
      }
    } catch (Exception ex) {
      System.out.println("error");
    }
  }
}
```

```
import java.io.*;
public class Album implements Serializable {
  private static int numberOfTracks = 5;
  private String title;
  private transient int currentTrack;

  public Album(String title, int currentTrack) {
    this.title = title;
    this.currentTrack = currentTrack;
  }

  public void readObject(ObjectInputStream in)
                         throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    currentTrack = 3;
  }

  @Override
  public String toString() {
    return title+" "+numberOfTracks +" "+currentTrack;
  }
}
```

```
import java.io.*;
public class RQ22 {
  public static void main(String[] args) {
    Album a = new Album("Songs", 2);
    try (ObjectOutputStream out =
             new ObjectOutputStream(new FileOutputStream("song.dat"))) {
      out.writeObject(a);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try (ObjectInputStream in =
             new ObjectInputStream(new FileInputStream("song.dat"))) {
      a = (Album)in.readObject();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    System.out.println(a);
  }
}
```


root_component
a
b
c
d
...
...
...
...

`/a/b/c`	on Unix-based platforms
`C:\a\b\c`	on Windows-based platforms

`c/d`	on Unix-based platforms
`c\d`	on Windows-based platforms

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

`static FileSystem getDefault()` Declared in `java.nio.file.FileSystems` `abstract String getSeparator()` Declared in `java.nio.file.FileSystem`

```
static Path of(String first, String... more)
static Path of(URI uri)
```

```
abstract Path getPath(String first, String... more)
```

```
static Path get(String first, String... more)
static Path get(URI uri)
```

```
FileSystem dfs = FileSystems.getDefault();     // Obtain the default file system.
String nameSeparator = dfs.getSeparator();     // The name separator for a path.

Path absPath1 = Path.of("/a/b/c");                         // (1) /a/b/c
Path absPath2 = Path.of(nameSeparator + "a" +              // (2) /a/b/c
                        nameSeparator + "b" +
                        nameSeparator + "c");
Path absPath3 = Path.of(nameSeparator, "a", "b", "c");     // (3) /a/b/c
System.out.println(absPath1.equals(absPath2) &&
                   absPath2.equals(absPath3));             // true
```

```
Path absPath4 = Path.of("C:\\a\\b\\c");                      // (4) C:\a\b\c
Path absPath5 = Path.of("C:", "a", "b", "c");                // (5) C:\a\b\c
Path relPath1 = Path.of("c", "d");                         //  c/d
```

```
String pathOfCurrDir = System.getProperty("user.dir");  // (1)
Path currDir = Path.of(pathOfCurrDir);                  // (2)
Path relPath = Path.of(pathOfCurrDir, "d");             // (3) <curr-dir-path>/d
Path absPath7 = Paths.get(nameSeparator, "a", "b", "c");
Path relPath3 = Paths.get("c", "d");
```

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

```
Path absPath6 = dfs.getPath(nameSeparator, "a", "b", "c");
Path relPath2 = dfs.getPath("c", "d");
```

```
File(String pathname)
Path toPath()
```

```
File file = new File(File.separator + "a" +
                     File.separator + "b" +
                     File.separator + "c");        // /a/b/c

// File --> Path, using the java.io.File.toPath() instance method
Path fileToPath = file.toPath();                   // /a/b/c

// Path --> File, using the java.nio.file.Path.toFile() default method.
File pathToFile = fileToPath.toFile();             // /a/b/c
```

```
file:///a/b/c/d                     // Scheme: file, to access a local file.
http://www.whatever.com             // Scheme: http, to access a remote website.
```

```
URI(String str) throws URISyntaxException
static URI create(String str)
```

```
// Create a URI object, using the URL.create(String str) static factory method.
URI uri1 = URI.create("file:///a/b/c/d");   // Local file.
```

```
// URI --> Path, using the Path.of(URI uri) static factory method.
Path uriToPath1 = Path.of(uri1);       // /a/b/c/d

// URI --> Path, using the Paths.get(URI uri) static factory method.
Path uriToPath2 = Paths.get(uri1);     // /a/b/c/d
```

```
// Path --> URI, using the Path.toUri() instance method.
URI pathToUri = uriToPath1.toUri();    // file:///a/b/c/d
```

```
Path subpath(int beginIndex, int endIndex)
```

```
boolean startsWith(Path other)
default boolean startsWith(String other)
```

```
boolean endsWith(Path other)
default boolean endsWith(String other)
```

```
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class QueryingPaths {

  public static void main(String[] args) {
    FileSystem dfs = FileSystems.getDefault();     // The default file system
    String nameSeparator = dfs.getSeparator();     // The name separator

    Path absPath = Path.of(nameSeparator, "a", "b", "c");                // (1)

    System.out.printf("toString(): %s%n",   absPath);                     // (2)
    System.out.printf("isAbsolute(): %s%n", absPath.isAbsolute());        // (3)
    System.out.printf("getFileSystem(): %s%n",
                      absPath.getFileSystem().getClass().getName());    // (4)

    System.out.println("\n***Accessing Name Elements:");                 // (5)
    System.out.printf("getFileName(): %s%n",  absPath.getFileName());
    System.out.printf("getParent(): %s%n",    absPath.getParent());
    System.out.printf("getRoot(): %s%n",      absPath.getRoot());
    System.out.printf("getNameCount(): %d%n", absPath.getNameCount());

    List<Path> pl = new ArrayList<>();
    absPath.forEach(p -> pl.add(p));
    System.out.printf("List of name elements: %s%n",  pl);

    System.out.printf("getName(0): %s%n",     absPath.getName(0));
    System.out.printf("subpath(0,2): %s%n",   absPath.subpath(0,2));

    System.out.println("\n***Path Prefix and Suffix:");                  // (6)
    System.out.printf("startsWith(\"%s\"): %s%n",
                      nameSeparator + "a",
                      absPath.startsWith(nameSeparator + "a"));
    System.out.printf("endsWith(\"b/c\"): %s%n",
                      absPath.endsWith("b/c"));
  }
}
```

```
toString(): /a/b/c
isAbsolute(): true
getFileSystem(): sun.nio.fs.MacOSXFileSystem

***Accessing Name Elements:
getFileName(): c
getParent(): /a/b
getRoot(): /
getNameCount(): 3
List of name elements: [a, b, c]
```

```
getName(0): a
subpath(0,2): a/b

***Path Prefix and Suffix:
startsWith("/a"): true
endsWith("b/c"): true
```

```
Path resolve(Path other)
default Path resolve(String other)
default Path resolveSibling(Path other)
default Path resolveSibling(String other)
```

```
Path toRealPath(LinkOption... options) throws IOException
```

```
System.out.println(absPath1.toAbsolutePath());      // (1) /a
System.out.println(path1.normalize());       // (1) a/b/c
System.out.println(absPath1.resolve(absPath2));       // (R1, C1)
```

```
Path anyPath = Path.of("/a/n/y");
Path emptyPath = Path.of("");
System.out.println(anyPath.resolve(emptyPath));       // /a/n/y
System.out.println(absPath1.revitalize(absPath2));      // (R1, C1)
```

```
Path p = Path.of("/a/b");
Path other = Path.of("/a/b/c/d");
Path q = p.relativize(other);                                // c/d
System.out.println(p.relativize(p.resolve(q)).equals(q));   // true
System.out.println(p.resolve(q).equals(other));              // true
```

```
try {
  Path somePath = Path.of("some/path");
  Path realPath = somePath.toRealPath(LinkOption.NOFOLLOW_LINKS);
  System.out.println(realPath);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
boolean equals(Object other)
```

```
Path p1 = Path.of("/", "a", "b", "c", "d");
Path p2 = Path.of("/", "a", "b");
Path p3 = Path.of("/", "a", "b", "c");
Path p4 = Path.of("a", "b");

// Sorting paths according to natural order:
List<Path> sortedPaths = Stream.of(p1, p2, p3, p4)
                               .sorted()
                               .toList();
System.out.println(sortedPaths);
// [/a/b, /a/b/c, /a/b/c/d, a/b]

// Comparing for lexicographical equality:
System.out.println(p2);                           // Absolute path: /a/b
System.out.println(p3.subpath(0, 2));             // Relative path: a/b
System.out.println(p2.equals(p3.subpath(0, 2))); // false
```

```
// Method header:
static boolean exists(Path path, LinkOption... options)  // Variable arity param.

// Method calls:
Path path = Path.of("alias");
boolean result1 = Files.exists(path);             // Follow symbolic links.
boolean result2 = Files.exists(path,
                    LinkOption.NOFOLLOW_LINKS);  // Do not follow symbolic links.
```

```
static boolean exists(Path path, LinkOption... options)
static boolean notExists(Path path, LinkOption... options)
```

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
System.out.println(Files.exists(path1));                          // true
System.out.println(Files.notExists(path1));                       // false

Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java");
System.out.println(Files.exists(path2));                          // true
System.out.println(Files.notExists(path2));                       // false

Path path3 = Path.of("project", "readme.txt");
System.out.println(Files.exists(path3));                          // false
System.out.println(Files.notExists(path3));                       // true
```

```
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

boolean result4 = Files.exists(target);                            // (1)
boolean result5 = Files.exists(symbLink);                          // (2)
boolean result6 = Files.exists(symbLink, LinkOption.NOFOLLOW_LINKS); // (3)

System.out.println("target: " + result4);                          // (1a) true
System.out.println("symbLink->target: " + result5);                // (2a) true
System.out.println("symbLink_NOFOLLOW_LINKS: " + result6);         // (3a) true
```

```
static boolean isSameFile(Path path1, Path path2) throws IOException
```

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java");

Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

System.out.println(Files.isSameFile(path1, path2));           // (1) true
System.out.println(Files.isSameFile(symbLink, target));       // (2) true
System.out.println(Files.isSameFile(path1, target));          // (3) false
System.out.println(Files.isSameFile(Path.of("Main.java"),
                                    Path.of("Main.java")));   // (4) true
System.out.println(Files.isSameFile(path1,
                           Path.of("Main.java")));  // (5) NoSuchFileException
```

```
static void delete(Path path) throws IOException
static boolean deleteIfExists(Path path) throws IOException
```

```
Path projDir  = Path.of("project");
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");
```

```
Files.delete(symbLink);                // Exists. Link deleted, not target.
Files.delete(Path.of("Main.java"));    // Does not exist: NoSuchFileException
```

```
System.out.println(Files.deleteIfExists(target));               // Exists.
                                                                //  Deleted: true
System.out.println(Files.deleteIfExists(Path.of("Main.java"))); // Does not
                                                                //   exist: false
```

```
Files.delete(projDir);                              // DirectoryNotEmptyException
System.out.println(Files.deleteIfExists(projDir)); // DirectoryNotEmptyException
```

```
static Path copy(Path source, Path destination, CopyOption... options)
                throws IOException
```

```
static long copy(InputStream in, Path destination, CopyOption... options)
                throws IOException
```

```
static long copy(Path source, OutputStream output) throws IOException
```

```
./project/archive/src/pkg/Main.java
```

```
Path source = Path.of("project", "src", "pkg", "Main.java");                // (1)
Path parentDestinationPath = Path.of("project", "archive", "src", "pkg");  // (2)
Path destination = parentDestinationPath.resolve(source.getFileName());    // (3)

Files.copy(source, destination);      // (4) OK. Destination file does not exist.
Files.copy(source, destination,       // (5) OK. Destination file replaced.
         StandardCopyOption.REPLACE_EXISTING);
Files.copy(source, destination);      // (6) FileAlreadyExistsException
```

```
Path srcDir  = Path.of("project", "src");
Path destDir = Path.of("project", "backup");
Files.copy(srcDir, destDir, StandardCopyOption.REPLACE_EXISTING);
```

```
Path srcFile = Path.of("project", "src", "pkg", "Main.java");
Path destDir = Path.of("project", "classes");
Files.copy(srcFile, destDir, StandardCopyOption.REPLACE_EXISTING);
```

```
String inputFileName  = "project/src/pkg/Util.java";
Path outputFilePath = Path.of("project", "backup", "Util.java");
```

```
try (var fis = new FileInputStream(inputFileName);
     var bis = new BufferedInputStream(fis)) {
  long bytesCopied = Files.copy(bis, outputFilePath,
                                StandardCopyOption.REPLACE_EXISTING);
  System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 103
}
```

```
Path inputFilePath  = Path.of("project", "backup", "Util.java");
String outputFileName = "project/archive/src/pkg/Util.java";
try (var fos = new FileOutputStream(outputFileName);
     var bos = new BufferedOutputStream(fos)) {
  long bytesCopied = Files.copy(inputFilePath, bos);
  System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 103
}
```

```
Files.copy(inputFilePath, System.out);     // Prints file content to standard out.
```

```
static Path move(Path source, Path destination, CopyOption... options)
        throws IOException
```

```
Path srcFile  = Path.of("project", "src",  "manifest.txt");
Path destFile = Path.of("project", "bkup", "manifest.txt");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING);
```

```
Path srcDir = Path.of("project", "bkup");
Path destDir = Path.of("project", "archive", "backup");  // Parent path exists.
Files.move(srcDir, destDir);
```

```
Path oldFileName  = Path.of("project", "backup", "Util.java");
Path newFileName  = Path.of("project", "backup", "UX.java");
Files.move(oldFileName, newFileName);
```

```
Path oldDirName  = Path.of("project", "backup");
Path newDirName  = Path.of("project", "bkup");
Files.move(oldDirName, newDirName);
```

```
Path srcFile = Path.of("project", "src", "pkg", "Util.java");
Path destFile = Path.of("project", "archive", "src", "pkg", "Util.java");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING,
                              StandardCopyOption.ATOMIC_MOVE);
```

```
static BufferedReader newBufferedReader(Path path) throws IOException
static BufferedReader newBufferedReader(Path path, Charset cs)
                                        throws IOException
```

```
static BufferedWriter newBufferedWriter(Path path, OpenOption... options)
                                    throws IOException
static BufferedWriter newBufferedWriter(Path path, Charset cs,
                      OpenOption... options) throws IOException
```

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ReadingWritingTextFiles {

  public static void main(String[] args) throws IOException {
    // List of strings:
    List<String> lines = List.of("Guess who got caught?", "Who?",
                                 "NullPointerException.",
                                 "Seriously?", "No. Finally.");
    // Text file:
    String filename = "project/linesOnly.txt";
    Path path = Path.of(filename);

    // Writing lines using buffered writer:                             (1)
    try (BufferedWriter writer =  Files.newBufferedWriter(path)) {   // (2)
      for(String str : lines) {
        writer.write(str);                  // Write a string.
        writer.newLine();                   // Terminate with a newline.
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    // Read lines using buffered reader:                                (3)
    lines = new ArrayList<>();
    try(BufferedReader reader= Files.newBufferedReader(path)) {      // (4)
      String line = null;
      while ((line = reader.readLine()) != null) {  // EOF when null is returned.
        lines.add(line);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    System.out.printf("Lines read from file \"%s\":%n%s%n", path, lines);
    // Write the list of strings in one operation:
    Files.write(path, lines);                                      // (5)

    // Write the joined lines in one operation:
    String joinedLines = String.join(System.lineSeparator(), lines);
    Files.writeString(path, joinedLines);                          // (6)

    // Read all contents into a String, including line separators:
    String allContent = Files.readString(path);                    // (7)
    System.out.printf("All lines read from file \"%s\":%n%s%n", path, allContent);

    // Read all lines into a list of String:
    lines = Files.readAllLines(path);                              // (8)
    System.out.printf("List of lines read from file \"%s\":%n%s%n", path, lines);
  }
}
```

```
Lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
All lines read from file "project/linesOnly.txt":
Guess who got caught?
Who?
NullPointerException.
Seriously?
No. Finally.
List of lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
```

```
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  OpenOption... options) throws IOException
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  Charset cs, OpenOption... options) throws IOException
```

```
static Path writeString(Path path, CharSequence csq, OpenOption... options)
          throws IOException
static Path writeString(Path path, CharSequence csq, Charset cs,
                    OpenOption... options) throws IOException
```

```
static String readString(Path path) throws IOException
static String readString(Path path, Charset cs) throws IOException
static List<String> readAllLines(Path path) throws IOException
static List<String> readAllLines(Path path, Charset cs) throws IOException
```

```
    Files.write(path, lines);                                      // (5)
```

```
String joinedLines = String.join(System.lineSeparator(), lines);
Files.writeString(path, joinedLines);                          // (6)
```

```
String allContent = Files.readString(path);                        // (7)
```

```
lines = Files.readAllLines(path);                              // (8)
```

```
static InputStream newInputStream(Path path, OpenOption... options)
                                  throws IOException
```

```
static OutputStream newOutputStream(Path path, OpenOption... options)
                    throws IOException
```

```
import java.io.*;
import java.nio.file.*;

public class ReadingWritingBytes {
  public static void main(String[] args) {
    // Source and destination files:
    Path srcPath  = Path.of("project", "source.dat");
    Path destPath = Path.of("project", "destination.dat");

    try (InputStream is = Files.newInputStream(srcPath);                        // (1)
         OutputStream os = Files.newOutputStream(destPath))  {
      byte[] buffer = new byte[1024];
      int length = 0;
      while((length = is.read(buffer, 0, buffer.length)) != -1) {
        os.write(buffer, 0, length);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    try {
      // Reads the file contents into an array of bytes:
      byte[] allBytes = Files.readAllBytes(srcPath);                          // (2)

      // Writes an array of bytes to a file:
      Files.write(destPath, allBytes);                                        // (3)
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

```
static byte[] readAllBytes(Path path) throws IOException
static Path write(Path path, byte[] bytes, OpenOption... options)
           throws IOException
```

```
byte[] allBytes = Files.readAllBytes(srcPath);                        // (2)
...
Files.write(destPath, allBytes);                                      // (3)
```

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

import static java.lang.System.out;
import static java.nio.file.attribute.PosixFilePermission.*;

public class IndividualFileAttributes {

  public static void main(String[] args) throws IOException {

    Path fPath = Path.of("project", "src", "pkg", "Main.java");
    out.println("File: " + fPath);

    out.println("Accessing Individual File Attributes:");
    out.println("size file (bytes): " + Files.size(fPath));                // (1)
    out.println("isDirectory:       " + Files.isDirectory(fPath));         // (2)
    out.println("isRegularFile:     " + Files.isRegularFile(fPath));       // (3)
    out.println("isSymbolicLink:    " + Files.isSymbolicLink(fPath));      // (4)
    out.println();

    out.println("isReadable:        " + Files.isReadable(fPath));          // (5)
    out.println("isWritable:        " + Files.isWritable(fPath));          // (6)
    out.println("isExecutable:      " + Files.isExecutable(fPath));        // (7)
    out.println("isHidden:          " + Files.isHidden(fPath));            // (8)
    out.println();

    out.println("getLastModifiedTime: " + Files.getLastModifiedTime(fPath));// (9)
    out.println("getOwner:            " + Files.getOwner(fPath));          // (10)
    out.println();

    // Get the POSIX file permissions for the directory entry:
    Set<PosixFilePermission> filePermissions
        = Files.getPosixFilePermissions(fPath);                            // (11)
    out.println("getPosixFilePermissions (set): " + filePermissions);      // (12)
    out.println("getPosixFilePermissions (string): "
              + PosixFilePermissions.toString(filePermissions));           // (13)

    // Get the group of the directory entry:
    out.println("getAttribute-group:  " + Files.getAttribute(fPath,        // (14)
                                                         "posix:group"));
    out.println();

    // Update last modified time for the directory entry.                     (15)
    long currentTime = System.currentTimeMillis();
    FileTime timestamp = FileTime.fromMillis(currentTime);
    Files.setLastModifiedTime(fPath, timestamp);

    // Set new owner for the directory entry.                                 (16)
    FileSystem fs = fPath.getFileSystem();   // File system that created the path.
    UserPrincipalLookupService upls
        = fs.getUserPrincipalLookupService();// Obtain service to look up user.
    UserPrincipal user = upls.lookupPrincipalByName("khalid"); // User lookup.
    Files.setOwner(fPath, user);                               // Set user.

    // Set POSIX file permissions for the directory entry:                    (17)
    Set<PosixFilePermission> newfilePermissions
        = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, GROUP_WRITE);   // (18a)
    //Set<PosixFilePermission> newfilePermissions
    //  = PosixFilePermissions.fromString("rw-rw----");                   // (18b)
    Files.setPosixFilePermissions(fPath, newfilePermissions);             // (19)
    filePermissions = Files.getPosixFilePermissions(fPath);
    out.println("getPosixFilePermissions (set): " + filePermissions);
    out.println("getPosixFilePermissions (string): "
          + PosixFilePermissions.toString(filePermissions));

    // Setting the value of a file attribute by its attribute name.
    Files.setAttribute(fPath, "lastAccessTime", timestamp);               // (20)
  }
}
```

```
File: project/src/pkg/Main.java
Accessing Individual File Attributes:
size file (bytes): 13
```

```
isDirectory:       false
isRegularFile:     true
isSymbolicLink:    false

isReadable:        true
isWritable:        true
isExecutable:      false
isHidden:          false

getLastModifiedTime: 2021-08-06T10:28:47.416033Z
getOwner:            khalid

getPosixFilePermissions (set): [OTHERS_READ, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-r--r--
getAttribute-group:  admin

getPosixFilePermissions (set): [GROUP_WRITE, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-rw----
```

```
static long size(Path path) throws IOException
static boolean isDirectory(Path path, LinkOption... options)
static boolean isRegularFile(Path path, LinkOption... options)
static boolean isSymbolicLink(Path path)
static boolean isReadable(Path path)
static boolean isWritable(Path path)
static boolean isExecutable(Path path)
static boolean isHidden(Path path) throws IOException
```

```
class java.nio.file.attribute.FileTime

static FileTime from(Instant instant)
static FileTime fromMillis(long value)
static FileTime getLastModifiedTime(Path path, LinkOption... options)
                                    throws IOException
static Path     setLastModifiedTime(Path path, FileTime time)
                                    throws IOException
```

```
static UserPrincipal getOwner(Path path, LinkOption... options)
static Path setOwner(Path path, UserPrincipal owner)
```

```
Set<PosixFilePermission> permSet1
      = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);
```

```
static Set<PosixFilePermission> fromString(String permStr)
static String toString(Set<PosixFilePermission> perms)
static FileAttribute<Set<PosixFilePermission>>
       asFileAttribute(Set<PosixFilePermission> perms)
```

```
// Get the POSIX file permissions for the directory entry:
Set<PosixFilePermission> filePermissions
    = Files.getPosixFilePermissions(fPath);                             // (11)
out.println("getPosixFilePermissions (set):    " + filePermissions);    // (12)
out.println("getPosixFilePermissions (string): "
           + PosixFilePermissions.toString(filePermissions));           // (13)
...
// Set POSIX file permissions for the directory entry:                    (17)
Set<PosixFilePermission> newfilePermissions
         = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);// (18a)
//Set<PosixFilePermission> newfilePermissions
//       = PosixFilePermissions.fromString("rw-r--r--");                // (18b)
Files.setPosixFilePermissions(fPath, newfilePermissions);               // (19)
```

```
static Set<PosixFilePermission>
       getPosixFilePermissions(Path path, LinkOption... options)
                                 throws IOException
```

```
static Path setPosixFilePermissions(Path path,
                                    Set<PosixFilePermission> perms)
                                    throws IOException
```

```
static Object getAttribute(Path path, String attribute,
                           LinkOption... options) throws IOException
static Path   setAttribute(Path path, String attribute, Object value,
                           LinkOption... options) throws IOException
```

```
BasicFileAttributes bfa = Files.readAttributes(path,                    // (1)
                                               BasicFileAttributes.class);
```

```
BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                                 BasicFileAttributeView.class);
```

```
static <A extends BasicFileAttributes> A
       readAttributes(Path path, Class<A> type, LinkOption... options)
       throws IOException
```

```
static <V extends FileAttributeView> V
      getFileAttributeView(Path path, Class<V> type, LinkOption... options)
```

```
static Map<String,Object> readAttributes(Path path, String attributes,
                                         LinkOption... options)
                                         throws IOException
```

```
interface java.nio.file.attribute.BasicFileAttributes

long size()
```

```
FileTime lastModifiedTime()
FileTime lastAccessTime()
FileTime creationTime()
```

```
// Declared in utility class FileUtils.
public static void printBasicFileAttributes(BasicFileAttributes bfa) {
  out.println("Printing basic file attributes:");
  out.println("lastModifiedTime: " + bfa.lastModifiedTime());
  out.println("lastAccessTime:   " + bfa.lastAccessTime());
  out.println("creationTime:     " + bfa.creationTime());

  out.println("size:             " + bfa.size());
  out.println("isDirectory:      " + bfa.isDirectory());
  out.println("isRegularFile:    " + bfa.isRegularFile());
  out.println("isSymbolicLink:   " + bfa.isSymbolicLink());
  out.println("isOther:          " + bfa.isOther());
  out.println();
}
```

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributes bfa = Files.readAttributes(path,                    // (1)
                                               BasicFileAttributes.class);
FileUtils.printBasicFileAttributes(bfa);                                // (2)
```

```
File: project/src/pkg/Main.java
Printing basic file attributes:
lastModifiedTime: 2021-07-23T10:15:34.854Z
lastAccessTime:   2021-07-23T10:16:33.166281Z
creationTime:     2021-07-20T23:03:58Z
size:             116
isDirectory:      false
```

```
isRegularFile:    true
isSymbolicLink:   false
isOther:          false
```

```
interface java.nio.file.attribute.PosixFileAttributes
          extends BasicFileAttributes

UserPrincipal owner()
GroupPrincipal group()
```

```
Set<PosixFilePermission> permissions()
```

```
// Declared in the utility class FileUtils.
public static void printPosixFileAttributes(PosixFileAttributes pfa) {
  out.println("Printing POSIX-specific file attributes:");
  UserPrincipal user = pfa.owner();
  GroupPrincipal group = pfa.group();
  Set<PosixFilePermission> permissions = pfa.permissions();
  String perms = PosixFilePermissions.toString(permissions);

  out.println("owner:            " + user);
  out.println("group:            " + group);
  out.println("permissions:      " + perms);
  out.println();
}
```

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributes pfa = Files.readAttributes(path,                   // (3)
                                              PosixFileAttributes.class);
FileUtils.printBasicFileAttributes(pfa);                               // (4)
FileUtils.printPosixFileAttributes(pfa);                               // (5)
```

```
File: project/src/pkg/Main.java
Printing basic file attributes:
...
Printing POSIX-specific file attributes:
owner:              javadude
group:              admin
permissions:        rw-r--r--
```

```
interface java.nio.file.attribute.DosFileAttributes
          extends BasicFileAttributes

boolean isReadOnly()
boolean isSystem()
boolean isArchive()
boolean isHidden()
```

```
interface BasicFileAttributeView extends FileAttributeView

String name()
```

```
BasicFileAttributes readAttributes()
```

```
void setTimes(FileTime lastModifiedTime,
              FileTime lastAccessTime,
              FileTime createTime)
```

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);

BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                               BasicFileAttributeView.class);
System.out.printf("Using view: %s%n", bfaView.name());

// Reading the basic set of file attributes:                               (7)
BasicFileAttributes bfa2 = bfaView.readAttributes();
FileUtils.printBasicFileAttributes(bfa2);
FileTime currentLastModifiedTime = bfa2.lastModifiedTime();

// Updating timestamp for last modified time using view:                   (8)
long newLMTinMillis = currentLastModifiedTime.toMillis() + 15*60*1000L;
FileTime newLastModifiedTime = FileTime.fromMillis(newLMTinMillis);
bfaView.setTimes(newLastModifiedTime, null, null);

// Reading the updated last modified time:                                 (9)
out.println("updated lastModifiedTime (incorrect): "
                                          + bfa2.lastModifiedTime());   // (10)
out.println("updated lastModifiedTime: "
                                    + Files.getLastModifiedTime(path)); // (11)
out.println("updated lastModifiedTime: " + Files.getAttribute(path,    // (12)
                                                    "basic:lastModifiedTime"));
```

```
File: project/src/pkg/Main.java
Using view: basic
Printing basic file attributes:
...
lastModifiedTime: 2021-07-26T15:15:46.813Z
...
updated lastModifiedTime (incorrect): 2021-07-26T15:15:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
```

```
interface PosixFileAttributeView
          extends BasicFileAttributeView, FileOwnerAttributeView
```

```
String name()
```

```
PosixFileAttributes readAttributes()
```

```
void setGroup(GroupPrincipal group) throws IOException
void setPermissions(Set<PosixFilePermission> perms)
```

```
interface java.nio.file.attribute.FileOwnerAttributeView
          extends FileAttributeView

UserPrincipal getOwner() throws IOException
void          setOwner(UserPrincipal owner) throws IOException
```

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);

PosixFileAttributeView pfaView = Files.getFileAttributeView(path,      // (13)
                                         PosixFileAttributeView.class);
System.out.printf("Using view: %s%n", pfaView.name());

// Reading the basic + POSIX set of file attributes:                   // (14)
PosixFileAttributes pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);

// Updating owner and group file attributes using view.                // (15)
FileSystem fs = path.getFileSystem();
UserPrincipalLookupService upls = fs.getUserPrincipalLookupService();
UserPrincipal newUser = upls.lookupPrincipalByName("javadude");
GroupPrincipal newGroup = upls.lookupPrincipalByGroupName("admin");
pfaView.setOwner(newUser);
pfaView.setGroup(newGroup);

//Updating file permissions using view.                                // (16)
Set<PosixFilePermission> newPerms = PosixFilePermissions.fromString("r--r--r--");
pfaView.setPermissions(newPerms);
```

```
//Updating last access time using view.                                  // (17)
FileTime currentAccessTime = pfa2.lastAccessTime();
long newLATinMillis = currentAccessTime.toMillis() + 10*60*1000L;
FileTime newLastAccessTime = FileTime.fromMillis(newLATinMillis);
pfaView.setTimes(null, newLastAccessTime, null);

// Reading the updated file attributes:                                  // (18)
pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);
```

```
interface DosFileAttributeView extends BasicFileAttributeView
```

```
String name()
```

```
DosFileAttributes readAttributes()
```

```
void setReadOnly(boolean value)
void setSystem(boolean value)
void setArchive(boolean value)
void setHidden(boolean value)
```

```
public static void printDirEntryInfo(Path path) throws IOException {
  String fmt = Files.isSymbolicLink(path)? "Symbolic link: %s%n":
      Files.isRegularFile(path)? "File: %s%n":
        Files.isDirectory(path)? "Directory: %s%n":
          "Directory entry: %s%n";
  out.printf(fmt, path);
  Set<PosixFilePermission> perms = Files.getPosixFilePermissions(path);
  String permStr = PosixFilePermissions.toString(perms);
  out.println(permStr);
}
```

```
static Path createFile(Path path, FileAttribute<?>... attrs)
                       throws IOException
```

```
static Path createTempFile(String prefix, String suffix,
                           FileAttribute<?>... attrs) throws IOException
static Path createTempFile(Path dir, String prefix, String suffix,
                           FileAttribute<?>... attrs) throws IOException
```

```
static Path createSymbolicLink(Path symbLink, Path target,
                              FileAttribute<?>... attrs) throws IOException
```

```
static Path readSymbolicLink(Path symbLink) throws IOException
```

```
Set<PosixFilePermission> filePerms = PosixFilePermissions.fromString("rw-r--r--");
FileAttribute<Set<PosixFilePermission>> fileAttr =
        PosixFilePermissions.asFileAttribute(perms);
```

```
try {
  Path regularFile  = Path.of("project", "docs", "readme.txt");
  Path createdFile1 = Files.createFile(regularFile, fileAttr);        // (1)
  FileUtils.printDirEntryInfo(createdFile1);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
try {
  // System property that defines the default temporary-file directory.      (1)
  String tmpdir = System.getProperty("java.io.tmpdir");
  System.out.println("Default temporary directory: " + tmpdir);

  // Create under the default temporary-file directory.                      (2)
  Path tmpFile1 = Files.createTempFile("events", ".log");
  FileUtils.printDirEntryInfo(tmpFile1);

  // Create under a specific directory:                                      (3)
  Path tmpFileDir = Path.of("project");
  Path tmpFile2 = Files.createTempFile(tmpFileDir, "proj_", ".dat", fileAttr);
  Path tmpFile3 = Files.createTempFile(tmpFileDir, "proj_", null,   fileAttr);
  Path tmpFile4 = Files.createTempFile(tmpFileDir, null,    ".dat", fileAttr);
  Path tmpFile5 = Files.createTempFile(tmpFileDir, null,    null,   fileAttr);
  FileUtils.printDirEntryInfo(tmpFile2);
  FileUtils.printDirEntryInfo(tmpFile3);
  FileUtils.printDirEntryInfo(tmpFile4);
  FileUtils.printDirEntryInfo(tmpFile5);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Default temporary directory: /var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/
File:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/events4720093907665196131.log
rw-------
File: project/proj_6062790522710209175.dat
rw-r--r--
File: project/proj_957015125593453845.tmp
rw-r--r--
File: project/3032983609251590109.dat
rw-r--r--
File: project/8205471872222375044.tmp
rw-r--r--
```

```
try {
  Path symbLinkPath  = Path.of(".", "readme_link");                     // (1)
  Path targetPath    = Path.of(".", "project", "backup", "readme.txt"); // (2)

  Path symbLink = Files.createSymbolicLink(symbLinkPath, targetPath);   // (3a)
//Path symbLink = Files.createSymbolicLink(symbLinkPath,
//                                         targetPath.toAbsolutePath());// (3b)
  Path target = Files.readSymbolicLink(symbLink);                       // (4)

  FileUtils.printDirEntryInfo(symbLink);
  FileUtils.printDirEntryInfo(target);
} catch (FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Symbolic link: ./readme_link
rw-r--r--
File: ./project/backup/readme.txt
rw-r--r--
```

```
static Path createDirectory(Path dir, FileAttribute<?>... attrs)
                        throws IOException
```

```
static Path createDirectories(Path dir, FileAttribute<?>... attrs)
            throws IOException
```

```
static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
                                throws IOException
static Path createTempDirectory(Path dir, String prefix,
                                FileAttribute<?>... attrs)
                                throws IOException
```

```
Set<PosixFilePermission> dPerms = PosixFilePermissions.fromString("rwxrwxrwx");
FileAttribute<Set<PosixFilePermission>> dirFileAttr =
        PosixFilePermissions.asFileAttribute(dPerms);
```

```
try {
  Path regularDir  = Path.of("project", "bin");
  Path createdDir = Files.createDirectory(regularDir);                    // (1)
  FileUtils.printDirEntryInfo(createdDir);

  if (Files.deleteIfExists(regularDir)) {                                 // (2)
    System.out.printf("Directory deleted: %s%n", regularDir);
  }
  Path newDir = Files.createDirectory(regularDir, dirFileAttr);           // (3)
  FileUtils.printDirEntryInfo(newDir);
  Files.setPosixFilePermissions(newDir, dPerms);                          // (4)
  FileUtils.printDirEntryInfo(newDir);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Directory: project/bin
rwxr-xr-
Directory deleted: project/bin
Directory: project/bin
rwxr-xr-x
Directory: project/bin
rwxrwxrwx
```

```
try {
  Path regularDir2  = Path.of("project", "branches", "maintenance", "versions");
  Path createdDir2 = Files.createDirectories(regularDir2, dirFileAttr);  // (5)
  FileUtils.printDirEntryInfo(createdDir2);
} catch (FileAlreadyExistsException fe) {                                // (6)
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Directory: project/branches/maintenance/versions
rwxrwxrwx
```

```
try  {
  // Create under the default temporary-file directory.                   (7)
  Path tmpDirPath1 = Files.createTempDirectory("log_dir", dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath1);

  // Create under a specific location:                                     (8)
  Path tmpDirLoc = Path.of("project");
  Path tmpDirPath2 = Files.createTempDirectory(tmpDirLoc, "log_dir", dirFileAttr);
  Path tmpDirPath3 = Files.createTempDirectory(tmpDirLoc, null, dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath2);
  FileUtils.printDirEntryInfo(tmpDirPath3);

  Files.setPosixFilePermissions(tmpDirPath3, dPerms);                   // (9)
  FileUtils.printDirEntryInfo(tmpDirPath3);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Directory:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/log_dir18136008118052819366
rwxr-xr-x
Directory: project/log_dir14908011762674796217
rwxr-xr-x
Directory: project/18428782809319921018
rwxr-xr-x
Directory: project/18428782809319921018
rwxrwxrwx
```

```
static Stream<String> lines(Path path) throws IOException
static Stream<String> lines(Path path, Charset cs) throws IOException
```

```
Path wordFile = Path.of(".", "project", "wordlist.txt");
System.out.println("Find palindromes, greater than length 2.");
try (Stream<String> stream = Files.lines(wordFile)){
  List<String> palindromes = stream
       .filter(str -> str.length() > 2)
       .filter(str -> str.equals(new StringBuilder(str).reverse().toString()))
       .toList();
  System.out.printf("List of palindromes:   %s%n", palindromes);
  System.out.printf("Number of palindromes: %s%n", palindromes.size());
} catch (IOException e) {
  e.printStackTrace();
}
```

```
Find palindromes, greater than length 2.
List of palindromes:   [aba, abba, aga, aha, ...]
Number of palindromes: 90
```

```
Path textFile = Path.of(".", "project", "linesOnly.txt");
try (Stream<String> stream = Files.lines(textFile)) {
  Map<Integer, Long> grpMap =
      stream.collect(Collectors.groupingBy(String::length,
                                           Collectors.counting()));
  System.out.println(grpMap);
} catch (IOException e) {
  e.printStackTrace();
}
```

Visited entry	Depth
./a	0
./a/x.txt	1
./a/b	1
./a/b/c	2
./a/b/c/z.txt	3
./a/b/c/dir_link	3
./a/b/d	2
./a/b/d/y.txt	3

```
Visited entry        Depth
./a                  0
./a/x.txt            1
./a/b                1
./a/b/d              2
./a/b/c              2
./a/b/c/z.txt        3
./a/b/c/dir_link     3
./a/b/d/y.txt        3
```

```
static Stream<Path> list(Path dir) throws IOException
```

```
Path dir = Path.of(".", "a");
System.out.printf("Immediate entries under directory \"%s\":%n", dir);
try(Stream<Path> stream = Files.list(dir)) {
  stream.forEach(System.out::println);
} catch (NotDirectoryException nde) {
nde.printStackTrace();
} catch (IOException ioe) {
ioe.printStackTrace();
}
```

```
Immediate entries under directory "./a":
./a/x.txt
./a/b
```

```
static Stream<Path> walk(Path start, FileVisitOption... options)
                          throws IOException
static Stream<Path> walk(Path start, int depth,
                          FileVisitOption... options)
                          throws IOException
```

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class WalkTheWalk {

  public static void main(String[] args) throws IOException {

    // Creating symbolic link.                                       // (1)
    try {
      Path targetPath = Path.of(".", "a", "b");
      Path symbLinkPath  = Path.of(".", "a", "b", "c", "dir_link");
      if (Files.notExists(symbLinkPath, LinkOption.NOFOLLOW_LINKS)) {
        Files.createSymbolicLink(symbLinkPath, targetPath.toAbsolutePath());
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return;
    }

    // Do the walk.                                                   // (2)
    Path start = Path.of(".", "a");
    int MAX_DEPTH = 4;
    for (int depth = 0; depth <= MAX_DEPTH; ++depth)  {                 // (3)
      try(Stream<Path> stream = Files.walk(start, depth,                // (4)
                                           FileVisitOption.FOLLOW_LINKS)) {
        System.out.println("Depth limit: " + depth);
        stream.forEach(System.out::println);
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
}
```

```
Depth: 0
./a
Depth: 1
./a
./a/x.txt
./a/b
Depth: 2
./a
./a/x.txt
./a/b
./a/b/c
./a/b/d
Depth: 3
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
./a/b/c/dir_link
./a/b/d
./a/b/d/y.txt
Depth: 4
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
Exception in thread "main" java.io.UncheckedIOException:
    java.nio.file.FileSystemLoopException: ./a/b/c/dir_link
...
```

```
/** Declared in FileUtils class.
 * Copy an entire directory.
 * @param sourceDir        Directory to copy.
 * @param destinationDir   Directory to which the source directory is copied.
 * @param options          Copy options for all entries.
 */
public static void copyEntireDirectory(Path sourceDir,
                                       Path destinationDir,
                                       CopyOption... options)  {
```

```
  try (Stream<Path> stream = Files.walk(sourceDir)) {                     // (1)
    stream.forEach(entry -> {
      Path relativeEntryPath = sourceDir.relativize(entry);              // (2)
      Path destination  = destinationDir.resolve(relativeEntryPath);    // (3)
      try {
        Files.copy(entry, destination, options);                         // (4)
      } catch (DirectoryNotEmptyException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    });
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

```
Source directory:      ./a/b
Current entry:         ./a/b/c/d
Relative entry path:   c/d
```

```
Destination directory:   ./x/y
Relative entry path:     c/d
Destination entry paths: ./x/y/c/d
```

```
CopyOption[] options = new CopyOption[] {
    StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES,
    LinkOption.NOFOLLOW_LINKS};
Path sourceDirectory      = Path.of(".", "a", "b");       //      ./a/b
Path destinationDirectory = Path.of(".", "x", "y");       // (5a) ./x/y
// Path destinationDirectory = Path.of(".", "x")
//              .resolve(sourceDirectory.getFileName());  // (5b) ./x/b
FileUtils.copyEntireDirectory(sourceDirectory, destinationDirectory, options);
```

```
static Stream<Path> find(Path start, int depth,
                         BiPredicate<Path,BasicFileAttributes> matcher,
                         FileVisitOption... options) throws IOException
```

```
System.out.println("Find regular files whose name ends with \".txt\""
                 + " and whose size is > 0:");
Path startEntry = Path.of(".", "a");
int depth = 5;
try (var pStream = Files.find(startEntry, depth,
                              (path, attrs) -> attrs.isRegularFile()       // (1)
                              && attrs.size() > 0
                              && path.toString().endsWith(".txt"))) {
   List<Path> pList = pStream.toList();
   System.out.println(pList);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

```
Find regular files whose name ends with ".txt" and whose size is > 0:
[./a/x.txt, ./a/b/c/z.txt, ./a/b/d/y.txt]
```

```
import java.io.IOException;
import java.nio.file.Path;

public final class Filing {
  public static void main (String[] args) throws IOException {
    Path file = Path.of("./document", "../book/../chapter1");
    System.out.println(file.toAbsolutePath());    // (1)
    System.out.println(file.toRealPath());        // (2)
    System.out.println(file.normalize());         // (3)
    System.out.println(file.toString());          // (4)
    System.out.println(file.getParent());         // (5)
  }
}
```

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class ListingFiles {
  public static void main(String[] args) throws IOException {
    Path dirPath = Path.of(".", "wrk");
    printFiles1(dirPath);
    printFiles2(dirPath);
    printFiles3(dirPath);
  }

  public static void printFiles1(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.list(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles2(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.walk(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles3(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.find(dirPath, Integer.MAX_VALUE,
                                         (p, a) -> true)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }
}
```

```
import java.nio.file.*;
public class RQ1 {
  public static void main(String[] args) {
    Path earth = Path.of("/", "planets", "earth");
    Path moonOrbit = earth.resolve(Path.of("moon", "orbit.param"));
    Path mars = earth.resolveSibling("mars");
    Path fromMarsToMoon = mars.relativize(moonOrbit);
    System.out.println(fromMarsToMoon);
  }
}
```

```
import java.nio.file.*;
public class RQ2 {
  public static void main(String[] args) {
    /* Assume current directory path is /planets. */
    Path path = Path.of("./mars/../earth").normalize().toAbsolutePath();
    System.out.println(path.getNameCount());
  }
}
```

```
import java.io.*;
import java.nio.file.*;
public class RQ3 {
  public static void main(String[] args) {
      try (var stream = Files.list(Path.of("/test")
                                       .toRealPath(LinkOption.NOFOLLOW_LINKS))) {
        stream.filter(p -> p.getFileName().toString().endsWith("txt"))
              .forEach(p -> System.out.println(p));
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
import java.io.*;
import java.nio.file.*;
public class RQ4 {
  public static void main(String[] args) {
    try (var stream = Files.walk(Path.of("/test")
                                     .toRealPath(LinkOption.NOFOLLOW_LINKS))) {
      stream.map(p -> p.getFileName().toString())
            .filter(p -> p.endsWith("txt"))
            .sorted()
            .forEach(System.out::println);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
import java.io.*;
import java.nio.file.*;
public class RQ6 {
  public static void main(String[] args) {
    try {
      Path p1 = Path.of("/users/joe/test/a.jpg");
      Path p2 = Path.of("/users").resolve(p1.getName(1).resolve("test/a.jpg"));
      Files.move(p1, p2);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
import java.nio.file.*;
public class RQ7 {
  public static void main(String[] args) {
    Path p1 = Path.of("/test");
    Path p2 = Path.of("store");
    System.out.println(p1.resolve(p2));
    System.out.println(p2.resolve(p1));
  }
}
```

```
import java.io.IOException;
import java.nio.file.*;
public class RQ8 {
  public static void main(String[] args) {
    try {
      Path source = Path.of("/test/readme.txt");
      Path destination = Path.of("/backup/readme_save.txt");
      Files.copy(source, destination);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
import java.io.IOException;
import java.nio.file.*;
public class RQ9 {
  public static void main(String[] args) {
    Path p = Path.of("/test/test.html");
    try (var stream = Files.lines(p)){
      String result = stream.filter(s->s.startsWith("<"))
                            .map(s->s.substring(s.indexOf(">"), s.indexOf(">")+1))
                            .reduce("", (s1,s2) -> s1+s2);
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

public class RQ10 {
  public static void main(String[] args) {
    Path directory = Path.of("/test/data");
    Path file = Path.of("/test/data/info.txt");
    try {
      Set<PosixFilePermission> permissions = new HashSet<>();
      permissions.add(PosixFilePermission.OWNER_READ);
      Files.setPosixFilePermissions(directory, permissions);

      permissions.add(PosixFilePermission.OWNER_WRITE);
      Files.setPosixFilePermissions(file, permissions);
      try (var stream = Files.walk(directory)) {
        stream.forEach(System.out::println);
      }
    } catch(IOException e) {
      System.out.println(e);
    }
  }
}
```

```
java.nio.file.AccessDeniedException:/test/data/info.txt
java.nio.file.AccessDeniedException:/test/data
```

```
import static java.nio.file.attribute.PosixFilePermission.*;
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.Set;

public class RQ11 {
  public static void main(String[] args) {
    try {
      Path file = Path.of("/test/data/info.txt");
      Set<PosixFilePermission> perms
              = PosixFilePermissions.fromString("---------");
      Files.setPosixFilePermissions(file, perms);

      PosixFileAttributeView pfaView = Files.getFileAttributeView(file,
          PosixFileAttributeView.class);
      PosixFileAttributes pfa = pfaView.readAttributes();

      perms = Set.of(OWNER_READ, GROUP_WRITE, OTHERS_READ);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.add(OWNER_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.remove(GROUP_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      System.out.println(perms);
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
```

```
[OWNER_READ, OWNER_WRITE, GROUP_WRITE, OTHERS_READ]
```

```
[OWNER_READ, OWNER_WRITE, OTHERS_READ]
```

```
import java.io.*;
import java.net.*;
import java.nio.file.*;
public class RQ12 {
  public static void main(String[] args) {
    try {
      URI uri = URI.create("file:///test/ora.html");
      Path p1 = Path.of(uri);
      Path p2 = Path.of("/test/ora.html");
      Files.copy(p1, p2);
      File file = p2.toFile();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
Thread thread = new Thread(
    () -> System.out.println("Harmonious threads create beautiful applications.")
);
thread.start();
```

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

```
final String getName()
final void setName(String name)
final void setDaemon(boolean flag)
final boolean isDaemon()
```

```
public class Counter implements Runnable {
  private int currentValue = 0;

  @Override
  public void run() {                             // (1) Thread entry point
    String threadName = Thread.currentThread().getName(); // (2)
    while (currentValue < 5) {
      System.out.printf("%s: old:%s new:%s%n",
                         threadName,              // (3) Print thread name,
                         this.currentValue,       //     old value,
                       ++this.currentValue);      //     new incremented value.
      try {
        Thread.sleep(500);                        // (4) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(threadName + " interrupted.");
      }
    }
    System.out.println("Exiting " + threadName);
  }
}
```

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();  // (5) main thread
    System.out.println("Method main() runs in thread " + threadName);

    // Create a Counter object:                            // (6)
    Counter counter = new Counter();

    // Create two threads with the same Counter:           // (7)
    Thread threadA = new Thread(counter, "Thread A");
    Thread threadB = new Thread(counter, "Thread B");

    // Start the two threads:                              // (8)
    System.out.println("Starting " + threadA.getName());
    threadA.start();
//  threadA.start();                            // (9) IllegalThreadStateException
    System.out.println("Starting " + threadB.getName());
    threadB.start();

    System.out.println("Exiting Thread " + threadName);    // (10)
  }
}
```

```
Method main() runs in thread main
Method main() runs in thread main
Starting Thread A
Starting Thread B
Exiting Thread main
Thread B: old:0 new:2
Thread A: old:0 new:1
```

```
(new Thread() {
   @Override
   public void run() {
     System.out.println("Harmonious threads create beautiful applications.")
   }
 }
).start();
```

```
public class Counter extends Thread {
  public int currentValue;
  public Counter() { this.currentValue = 0; }
  public int getValue() { return this.currentValue; }

  @Override
  public void run() {                          // (1) Override from superclass.
    while (this.currentValue < 5) {
      System.out.printf("%s: %s%n",
                    super.getName(),           // (2) Print thread name,
                    this.currentValue++);      //     current value, and increment.
      try {
        Thread.sleep(500);                     // (3) Current thread sleeps.
      } catch (InterruptedException e) {
        System.out.println(super.getName() + " interrupted.");
      }
    }
    System.out.println("Exiting " + super.getName());
  }
}
```

```
public class Client {
  public static void main(String[] args) {
    String threadName = Thread.currentThread().getName();              // (4)
    System.out.println("Method main() runs in thread " + threadName);

    // Create two Counter objects that extend the Thread class:            (5)
    Counter counterA = new Counter();
    Counter counterB = new Counter();

    // Set the names for the two threads:                                  (6)
    counterA.setName("Counter A");
    counterB.setName("Counter B");
    // Mark the threads as daemon threads:                                   (7)
//  counterA.setDaemon(true);
//  counterB.setDaemon(true);

    // Start the two threads:                                             // (8)
    System.out.println("Starting " + counterA.getName());
    counterA.start();
    System.out.println("Starting " + counterB.getName());
    counterB.start();

    System.out.println("Exiting " + threadName);
  }
}
```

```
Method main() runs in thread main
Starting Counter A
Counter A: 0
Starting Counter B
Exiting main
Counter B: 0
Counter A: 1
Counter B: 1
Counter B: 2
Counter A: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exiting Counter B
Exiting Counter A
```

```
counterA.setDaemon(true);
counterB.setDaemon(true);
```

```
Method main() runs in thread main
Starting Counter A
Starting Counter B
Counter A: 0
Counter B: 0
Exiting main
```

```
public class MyClass extends Thread {
  public MyClass(String s) { msg = s; }
  String msg;
  public void run() {
    System.out.println(msg);
  }

  public static void main(String[] args) {
    new MyClass("Hello");
    new MyClass("World");
  }
}
```

```
class R1 implements Runnable {
  public void run() {
    System.out.print(Thread.currentThread().getName());
  }
}
public class R2 implements Runnable {
  public void run() {
    new Thread(new R1(),"|R1a|").run();
    new Thread(new R1(),"|R1b|").start();
    System.out.print(Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    new Thread(new R2(),"|R2|").start();
  }
}
```

```
public class Threader extends Thread {
  Threader(String name) {
    super(name);
  }
  public void run() throws IllegalThreadStateException {
    System.out.println(Thread.currentThread().getName());
    throw new IllegalThreadStateException();
  }
```

```
static boolean holdsLock(Object obj)
```

```
final int getPriority()
final void setPriority(int newPriority)
```

```
static void sleep(long millisec) throws InterruptedException
static void sleep(long millis, int nanos) throws InterruptedException
```

```
final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException
```

```
static Map<Thread, StackTraceElement[]> getAllStackTraces()
```

```
public class ThreadStates {

  private static Thread t1 = new Thread("T1") {    // (1)
    @Override public void run() {
      try {
        sleep(1);                                  // (2)
        for (int i = 10000; i > 0; i--) {;}        // (3)
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  };

  public static void main(String[] args) {
    System.out.println(t1.getState());              // (4)
    t1.start();
    while (true) {                                  // (5)
      Thread.State state = t1.getState();
      System.out.println(state);
      if (state == Thread.State.TERMINATED) {
        break;
      }
    }
  }
}
```

```
Thread myThread = new Thread(() ->
    System.out.println(Thread.currentThread() + ": Don't mess with my priority!")
);
System.out.println(myThread);
myThread.setPriority(Math.min(Thread.MAX_PRIORITY, myThread.getPriority() + 1));
myThread.start();
```

```
Thread[Thread-0,5,main]
Thread[Thread-0,6,main]: Don't mess with my priority!
```

```
public void run() {
  try {
    while (!done()) {
      doLittleBitMore();
      Thread.yield();                 // Current thread yields.
    }
  } catch (InterruptedException ie) { // Clean up if the thread is interrupted.
    doCleaningUp();
  }
}
```

```
class CounterX {
  private int counter = 0;

  public synchronized void increment() {                            // (1a)
//public void increment() {                                         // (1b)
    System.out.println(Thread.currentThread().getName()
                     + ": old:" + counter + " new:" + ++counter);
  }
}
```

```
public class SynchronizedMethodDemo {
  public static void main(String[] args) {
    CounterX counter = new CounterX();                              // (2)
    Runnable r = () -> {                                            // (3)
      for (int i = 0; i < 5; i++) {
        counter.increment();
      }
      System.out.println("Exiting " + Thread.currentThread().getName());
    };
    new Thread(r).start();                                          // (4)
    new Thread(r).start();                                          // (5)
    System.out.println("Exiting thread " + Thread.currentThread().getName());
  }
}
```

`synchronized` (*object_reference_expression*) { *code_block* }

```
public void increment() {
  synchronized(this) {                                // Synchronized statement
    System.out.println(Thread.currentThread().getName()
        + ": current: " + counter + " new:" + ++counter);
  }
}
```

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
    synchronized (account) {       // (1) synchronized statement
      account.update();            // (2)
    }
  }
}
```

```
class Outer {                        // (1) Top-level Class
  private double count;              // (2)

  protected class Inner {            // (3) Non-static member Class
    public void incr() {             // (4)
      synchronized(Outer.this) {     // (5) Synchronized statement on Outer object
        ++count;                     // (6)
      }
    }
  }
}
```

```
synchronized (<class_name>.class) { <code_block> }
```

```
static void classAction() {
  synchronized (A.class) {       // Synchronized statement on class A
```

```
    // ...
  }
}
```

```
public class InterruptHandling {
  public static void main(String[] args) throws InterruptedException {
    Thread worker = new Thread(task, "worker");                     // (1)
    worker.start();
    while (true) {                                                   // (2)
      if ((int)(Math.random()*100) % 2 == 0) {
        worker.interrupt();
        break;
      }
      Thread.sleep(2);
    }
  }
  private static Runnable task = () -> {                              // (3)
    Thread ct = Thread.currentThread();
    String threadName = ct.getName();
    while (true) {                                                    // (4)
      System.out.println(threadName + " performing task");

      if (ct.isInterrupted()) {                                       // (5)
        System.out.println(threadName + ": interrupted flag is "
                         + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }

      try {                                                            // (6)
        Thread.sleep(2);
      } catch (InterruptedException e) {                               // (7)
        System.out.println(threadName + " caught " + e);
        System.out.println(threadName + ": interrupted flag is "
                         + ct.isInterrupted());
        System.out.println(threadName + " terminating");
        return;
      }
    }
  };
}
```

```
worker performing task
worker: interrupted flag is true
worker terminating
worker performing task
worker performing task
worker caught java.lang.InterruptedException: sleep interrupted
worker: interrupted flag is false
worker terminating
```

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

```
while (conditionNotSatisfied()) {
  ...
  wait();
  ...
}
// Proceed ...
```

```
class MessageDisplay {
  private String message;

  public synchronized void displayMessage() {
    String threadName = Thread.currentThread().getName();
    while (this.message == null) {                          // No message?
      try {
        wait();                                             // (1)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println(threadName + ": " + this.message);   // Display message.
    this.message = null;                                    // Remove message.
    notifyAll();                                            // (2)
  }

  public synchronized void setMessage(String message) {
    String threadName = Thread.currentThread().getName();
    while (this.message != null) {                          // Message present?
      try {
        wait();                                             // (3)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.message = message;                                 // Set new message.
    System.out.println(threadName + ": Message set is " + this.message);
    notifyAll();                                            // (4)
  }
}
```

```
public class WaitNotifyScenario {
  public static void main(String[] args) throws InterruptedException {
    MessageDisplay md = new MessageDisplay();
    Thread t1 = new Thread(() -> { while (true) md.setMessage("Hi!"); }, "t1");
    t1.setDaemon(true);
    t1.start();
    Thread t2 = new Thread(() -> { while (true) md.setMessage("Howdy!"); }, "t2");
    t2.setDaemon(true);
    t2.start();
    Thread t3 = new Thread(() -> { while (true) md.displayMessage(); }, "t3");
    t3.setDaemon(true);
    t3.start();
    Thread t4 = new Thread(() -> { while (true) md.displayMessage(); }, "t4");
    t4.setDaemon(true);
    t4.start();
    Thread.sleep(5);
    System.out.println("Exit from main.");
  }
}
```

```
class Counter extends Thread { /* See Example 22.2, p. 1376. */ }
```

```
public class AnotherClient {
  public static void main(String[] args) {

    // Create two Counter threads, set their names, and start them:    // (4)
    Counter counterA = new Counter();
    Counter counterB = new Counter();
    counterA.setName("counterA");
    counterB.setName("counterB");
    counterA.start();
    counterB.start();

    try {
      System.out.println("Wait for the child threads to finish.");
      counterA.join();                                                 // (5)
      if (!counterA.isAlive()) {                                       // (6)
        System.out.println("Counter A not alive.");
      }
      counterB.join();                                                 // (7)
      if (!counterB.isAlive()) {
        System.out.println("Counter B not alive.");
      }
    } catch (InterruptedException ie) {
      System.out.println("main thread interrupted.");
    }
    System.out.println("Exiting from main thread.");
  }
}
```

```
Wait for the child threads to finish.
counterB: 0
counterA: 0
counterA: 1
counterB: 1
counterA: 2
counterB: 2
counterA: 3
counterB: 3
counterA: 4
counterB: 4
Exiting counterB
Exiting counterA
Counter A not alive.
Counter B not alive.
Exiting from main thread.
```

```
public class Worker implements Runnable {                           // (1)
  private Thread theThread;                                          // (2)

  public void kickStart() {                                          // (3)
    if (theThread == null) {
      theThread = new Thread(this);
      theThread.start();
    }
  }

  public void terminate() {                                          // (4)
    theThread = null;
  }

  @Override
  public void run() {                                                // (5)
    while (theThread == Thread.currentThread()) {                    // (6)
      System.out.println("Going around in loops.");
    }
  }
}
```

```
public class Controller {
  public static void main(String[] args) {                        // (7)
    Worker worker = new Worker();                                 // (8)
    System.out.println("Start the worker.");
    worker.kickStart();                                           // (9)
    try {
      Thread.sleep(2);                                            // (10)
    } catch(InterruptedException ie) {
      ie.printStackTrace();
    }
    System.out.println("Stop the worker.");
    worker.terminate();                                           // (11)
  }
}
```

```
public class DeadLockDanger {
  public static void main(String[] args) {
    String o1 = "o1 " ;                                         // (1)
    String o2 = "o2 ";                                          // (2)

    Thread t1 = (new Thread("t1") {                             // (3)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o1) {                                    // (4)
            System.out.println(threadName + " acquired " + o1);
            synchronized(o2) {                                  // (5)
              System.out.println(threadName + ": " + o1 + o2);
      }}}}});
    Thread t2 = (new Thread("t2") {                                     // (6)
      @Override
      public void run() {
        String threadName = Thread.currentThread().getName();
        while (true) {
          synchronized(o2) {                                            // (7)
            System.out.println(threadName + " acquired " + o2);
            synchronized(o1) {                                          // (8)
              System.out.println(threadName + ": " + o2 + o1);
      }}}}});

    t1.start();                                                         // (9)
    t2.start();                                                         // (10)
    System.out.println("Exiting main.");                                // (11)
  }
}
```

```
public class Customer {                                                  // (1)
  private boolean paymentMade = false;                                   // (2)

  public void makePaymentTo(Seller seller) {                             // (3)
    while (!seller.hasShipped()) {
      System.out.println("Customer: waiting for shipment from seller");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Customer: payment made");
    this.paymentMade = true;                                             // (4)
  }

  public boolean hasPaid() {
    return this.paymentMade;
  }
}
```

```
public class Seller {                                                    // (5)
  private boolean itemShipped = false;                                   // (6)

  public void shipTo(Customer customer) {                                // (7)
    while (!customer.hasPaid()) {
      System.out.println("Seller: waiting for payment from customer");
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Seller: item shipped");
    this.itemShipped = true;                                            // (8)
  }

  public boolean hasShipped() {
    return this.itemShipped;
  }
}
```

```
public class LivelockShipment {
  public static void main(String[] args) {
    Customer customer = new Customer();
    Seller seller = new Seller();

    new Thread(() -> customer.makePaymentTo(seller)).start();             // (9)
    new Thread(() -> seller.shipTo(customer)).start();                    // (10)
  }
}
```

```
...
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
Seller: waiting for payment from customer
Customer: waiting for shipment from seller
Seller: waiting for payment from customer
...
```

```
public class Hole {                                                    // (1)
  public synchronized void dig() {
    String threadName = Thread.currentThread().getName();
    while (true) {                                                     // (2)
      System.out.println(threadName + " digging the hole.");
//    try {                                                            // (3)
//    wait(1);
//    } catch (InterruptedException ex) {
//      ex.printStackTrace();
//    }
    }
  }
}
```

```
public class Diggers {
  public static void main(String[] args) {
    Hole hole = new Hole();                                                  // (4)
    for (int i = 0; i < 5; i++) {                                            // (5)
      new Thread(() -> hole.dig()).start();
    }
  }
}
```

```
  int counter = 0;

// Thread A                                   // Thread B
...                                           ...
counter = 1;                                  ...
System.out.println("Thread A:" + counter);   System.out.println("Thread B:" + counter);
...                                           ...
```

```
public class ThreadedPrint {
  static Thread makeThread(String id, boolean daemon) {
    Thread t = new Thread(() -> System.out.println(id), id);
    t.setDaemon(daemon);
    t.start();
    return t;
  }

  public static void main(String[] args) {
    Thread a = makeThread("A", false);
    Thread b = makeThread("B", true);
    System.out.println("End");
  }
}
```

```
public class Smiley extends Thread {

  public void run() {                                              // (1)
    while (true) {                                                 // (2)
      try {                                                        // (3)
          System.out.print(":");                                   // (4)
          sleep(100);                                              // (5)
          System.out.print("-");                                   // (6)
          sleep(100);                                              // (7)
          System.out.println(")");                                 // (8)
          sleep(100);                                              // (9)
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  public static void main(String[] args) {
    new Smiley().start();
    new Smiley().start();
  }
}
```

```
public class Joining {
  static Thread createThread(final int i, final Thread t1) {
    Thread t2 = new Thread() {
      public void run() {
        System.out.println(i+1);
        try {
          t1.join();
        } catch (InterruptedException ie) {
        }
        System.out.println(i+2);
      }
    };
    System.out.println(i+3);
    t2.start();
    System.out.println(i+4);
    return t2;
  }

  public static void main(String[] args) {
    createThread(10, createThread(20, Thread.currentThread()));
  }
}
```

```
public class Tank {
  private boolean isEmpty = true;

  public synchronized void emptying() {
    pause(true);
    isEmpty = !isEmpty;
    System.out.println("emptying");
    notify();
  }

  public synchronized void filling() {
    pause(false);
    isEmpty = !isEmpty;
    System.out.println("filling");
    notify();
  }

  private void pause(boolean flag) {
    while (flag ? isEmpty : !isEmpty) {
      try {
        wait();
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread().getName() + " interrupted.");
      }
    }
  }

  public static void main(String[] args) {
    final Tank token = new Tank();
    new Thread(() -> {for(;;) token.emptying();}, "A").start();
    new Thread(() -> {for(;;) token.filling();}, "B").start();
  }
}
```

```
boolean allTerminated = executor.awaitTermination(2, TimeUnit.SECONDS);
```

```
TimeUnit.SECONDS.sleep(1);               // 1 second.
TimeUnit.MILLISECONDS.sleep(1000);       // 1000 milliseconds.
Thread.sleep(1000);                      // 1000 milliseconds.
```

```
void sleep(long timeout)                       throws InterruptedException
void timedJoin(Thread thread, long timeout) throws InterruptedException
void timedWait(Object obj, long timeout)     throws InterruptedException
```

```
int diceValue = ThreadLocalRandom.current().nextInt(1, 7);   // [1, 6]
```

```
long count = ThreadLocalRandom.current()
                              .ints(100, 1, 7).filter(i -> i == 6).count();
```

```
static ThreadLocalRandom current()
```

```
int nextInt(int bound)
int nextInt(int origin, int bound)
```

```
IntStream ints()
IntStream ints(long streamSize)
IntStream ints(int randomNumberOrigin, int randomNumberBound)
IntStream ints(long streamSize, int randomNumberOrigin,
               int randomNumberBound)
```

```
class SimpleExecutor implements Executor {
  @Override
  public void execute(Runnable task) {

    new Thread(task).start();         // (1a) Asynchronous call
//  task.run();                       // (1b) Synchronous call. Not recommended.
  }
}
// Client code
SimpleExecutor executor = new SimpleExecutor();
Runnable task = () -> System.out.println("Executing task ...");
executor.execute(task);
```

```
    new Thread(task).start();          // (1a) Asynchronous call
//  task.run();                        // (1b) Synchronous call. Not recommended.
  }
}
// Client code
SimpleExecutor executor = new SimpleExecutor();
Runnable task = () -> System.out.println("Executing task ...");
executor.execute(task);
```

```
newWorkStealingPool()
newWorkStealingPool(int parallelism)
```

`newSingleThreadScheduledExecutor()`

```
boolean awaitTermination(long timeout, TimeUnit unit)
                                throws InterruptedException
```

```
package executors;
import java.util.concurrent.*;

public class ExecutorLifecycle {

  // Task: dice roll.
  private static final Runnable diceRoll = () -> {                         // (1)
    int diceValue = ThreadLocalRandom.current().nextInt(1, 7);          // [1, 6]

    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + " => dice value: " + diceValue);
    try {
      TimeUnit.MILLISECONDS.sleep(100);
    } catch (InterruptedException e) {
      System.out.println(threadName + ": " + e);
    }
  };

  public static void main(String[] args) {
    System.out.printf("%50s   %s%n", "isShutdown()", "isTerminated()");
    // Create the executor service:
    ExecutorService es = Executors.newFixedThreadPool(3);                 // (2)
    try {                                                                  // (3)
      checkStates(es, "Before execute() at (4): ");
      // Submit tasks:
      es.execute(diceRoll);                                                // (4)
      es.execute(diceRoll);
      es.execute(diceRoll);
      checkStates(es, "After execute() at (4): ");
    } finally {                                                            // (5)
      // Shut down the executor service:
      checkStates(es, "Before shutdown() at (6a): ");
      es.shutdown();                                                       // (6a)
      checkStates(es, "After shutdown() at (6a): ");

//    checkStates(es, "Before shutdownNow() at (6b): ");
//    es.shutdownNow();                                                    // (6b)
//    checkStates(es, "After shutdownNow() at (6b): ");
    }
    // Second phase of shutdown:
//  awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                          // (7a)
//  awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                     // (7b)
  }

  private static void checkStates(ExecutorService es, String msg) {        // (8)
    System.out.printf("%-40s %-5s           %-5s%n",
        msg, es.isShutdown(), es.isTerminated());
  }

  private static void awaitAndShutdownNow(                                 // (9)
      ExecutorService es, int timeout, TimeUnit timeunit) {
    try {
      // Timed wait for tasks to complete execution:
      if (!es.awaitTermination(timeout, timeunit)) {                       // (10)
        // Attempt to cancel any uncompleted and waiting tasks:
        checkStates(es, "Before shutdownNow() at (11): " );
        es.shutdownNow();                                                  // (11)
        checkStates(es, "After shutdownNow() at (11): " );
        // Timed wait for tasks to be cancelled:
        while (!es.awaitTermination(timeout, timeunit)) {                  // (12)
          System.out.println("All tasks not yet completed at (12).");
        }
      }
      checkStates(es, "After awaitTermination() at (10): " );
    } catch (InterruptedException ie) {                                    // (13)
      // Attempt to cancel any uncompleted and waiting tasks:
      es.shutdownNow();                                                    // (14)
      // Reinstate the interrupt status.
      Thread.currentThread().interrupt();
      checkStates(es, "After interruption: ");
    }
  }
}
```

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
Before shutdown() at (6a):               false           false
pool-1-thread-3 => dice value: 6
pool-1-thread-1 => dice value: 2
pool-1-thread-2 => dice value: 1
After shutdown() at (6a):                true            false
```

```
pool-1-thread-3 => dice value: 6
pool-1-thread-1 => dice value: 2
pool-1-thread-2 => dice value: 1
After shutdown() at (6a):                   true             false
```

```
                                      isShutdown()   isTerminated()
Before execute() at (4):                 false           false
After execute() at (4):                  false           false
pool-1-thread-3 => dice value: 4
pool-1-thread-2 => dice value: 5
pool-1-thread-1 => dice value: 5
Before shutdownNow() at (6b):            false           false
After shutdownNow() at (6b):             true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

```
ExecutorService es = Executors.newFixedThreadPool(3);
```

```
es.execute(diceRoll);                                                       // (4)
es.execute(diceRoll);
es.execute(diceRoll);
```

```
// Create the executor service:
ExecutorService es = Executors.newFixedThreadPool(3);
try {                                                                     // (3)
  // Submit tasks:
  es.execute(diceRoll);                                                   // (4)
  es.execute(diceRoll);
  es.execute(diceRoll);
} finally {                                                               // (5)
  // Shut down the executor service:
  es.shutdown();                                                          // (6a)

//es.shutdownNow();                                                       // (6b)
}
```

```
awaitAndShutdownNow(es, 2, TimeUnit.SECONDS);                         // (7a)
```

```
                                       isShutdown()   isTerminated()
Before execute() at (4):                  false            false
After execute() at (4):                   false            false
Before shutdown() at (6a):                false            false
pool-1-thread-2 => dice value: 4
pool-1-thread-3 => dice value: 1
pool-1-thread-1 => dice value: 2
After shutdown() at (6a):                 true             false
After awaitTermination() at (10):         true             true
```

```
awaitAndShutdownNow(es, 1, TimeUnit.MILLISECONDS);                    // (7b)
```

```
                                             isShutdown()   isTerminated()
Before execute() at (4):                        false           false
After execute() at (4):                         false           false
Before shutdown() at (6a):                      false           false
pool-1-thread-2 => dice value: 3
pool-1-thread-1 => dice value: 1
pool-1-thread-3 => dice value: 1
After shutdown() at (6a):                       true            false
Before shutdownNow() at (11):                   true            false
After shutdownNow() at (11):                    true            false
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
After awaitTermination() at (10):               true            true
```

```
Random rng = new Random();
Callable<Integer> dice = () -> rng.nextInt(1,7);  // (1)
try {
  Integer diceValue = dice.call();     // (2)
  System.out.println(diceValue);       // Prints a value in the interval [1,6].
} catch (Exception ex) {
  ex.printStackTrace();
}
```

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
      throws InterruptedException, ExecutionException, TimeoutException
```

```
boolean cancel(boolean mayInterruptIfRunning)
```

```
Future<?> submit(Runnable task)
<V> Future<V> submit(Runnable task, V result)
<V> Future<V> submit(Callable<V> task)
```

```
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks)
                              throws InterruptedException
<V> List<Future<V>> invokeAll(Collection<? extends Callable<V>> tasks,
                              long timeout, TimeUnit unit)
                              throws InterruptedException
```

```
<V> V invokeAny(Collection<? extends Callable<V>> tasks)
                throws InterruptedException, ExecutionException
<V> V invokeAny(Collection<? extends Callable<V>> tasks,
                long timeout, TimeUnit unit)
                throws InterruptedException, ExecutionException,
                       TimeoutException
```

```
package executors;
import java.util.List;
import java.util.concurrent.*;
import java.util.stream.Collectors;

public class TaskExecution {

  public static void main(String[] args) {
    // Runnable:
    Runnable printDiceValue = () ->                                    // (1)
      System.out.println("Execution of Runnable: "
                      + ThreadLocalRandom.current().nextInt(1,7));

    // Callable<V>:
    Callable<Integer> diceRoll =                                       // (2)
        () -> ThreadLocalRandom.current().nextInt(1,7);

    // Executor service:
    ExecutorService exs = Executors.newFixedThreadPool(3);             // (3)
    try {
      // Executing Runnable in a thread:
      Future<?> rfuture = exs.submit(printDiceValue);                  // (4)
      Object result = rfuture.get();                                   // (5)
      System.out.println("Result of Runnable: " + result);

      // Executing Callable<V> in a thread:
      Future<Integer> cfuture = exs.submit(diceRoll);                  // (6)
      Integer diceValue = cfuture.get();                               // (7)
      System.out.println("Result of Callable: " + diceValue);

      // Executing bulk tasks:
      List<Callable<Integer>> callables                                // (8)
          = List.of(diceRoll, diceRoll, diceRoll);
      List<Future<Integer>> allFutures = exs.invokeAll(callables);         // (9)
      List<Integer> resultList = allFutures.stream()                       // (10)
          .map(future -> {
            try {
              return future.get();
            } catch(InterruptedException | ExecutionException ie) {
              throw new IllegalStateException(ie);
            }
          })
          .toList();
      System.out.println("Result of invokeAll(): " + resultList);

      Integer anyResult = exs.invokeAny(callables);                        // (11)
      System.out.println("Result of invokeAny(): " + anyResult);
    } catch(InterruptedException | ExecutionException ie) {
      ie.printStackTrace();
    } finally {
      exs.shutdown();
    }
  }
}
```

```
Execution of Runnable: 4
Result of Runnable: null
Result of Callable: 2
Result of invokeAll(): [1, 3, 6]
Result of invokeAny(): 3
```

```
ScheduledFuture<?> schedule(Runnable task, long delay, TimeUnit unit)
```

```
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay,
                                   TimeUnit unit)
```

```
package executors;
import java.util.concurrent.*;

public class ScheduledOneShotTasks {
  public static void main(String[] args) {
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);  // (1)
    try {
      // Schedule a one-shot Runnable:
      Runnable runnableTask                                              // (2)
          = () -> System.out.println("I am a one-shot Runnable task.");
      ScheduledFuture<?> scheduledRunnableTaskFuture                     // (3)
          = ses.schedule(runnableTask, 500, TimeUnit.MILLISECONDS);
      System.out.println("Runnable result: " + scheduledRunnableTaskFuture.get());

      // Schedule a one-shot Callable<String>:
      Callable<String> callableTask
          = () -> "I am a one-shot Callable task.";                      // (4)
      ScheduledFuture<String> scheduledCallableTaskFuture
          = ses.schedule(callableTask, 1, TimeUnit.SECONDS);             // (5)
      System.out.println("Callable result: " + scheduledCallableTaskFuture.get());
    } catch (InterruptedException | ExecutionException exc) {
      exc.printStackTrace();
    } finally {
      ses.shutdown();
    }
  }
}
```

```
I am a one-shot Runnable task.
Runnable result: null
Callable result: I am a one-shot Callable task.
```

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay,
                                          long delay, TimeUnit unit)
ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay,
                                       long period, TimeUnit unit)
```

```
package executors;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.*;

public class ScheduledPeriodicTasks {

  private static Runnable task = () -> {                                // (1)
    printTimestamp(" Start:  I am on it!");
    try {
      TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    printTimestamp(" Finish: I am not on it!");
  };

  private static void printTimestamp(String msg) {                      // (2)
    String threadName = Thread.currentThread().getName();
    String ts = LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm:ss"));
    System.out.println(threadName + ": " + ts + msg);
  }

  public static void main(String[] args) {
    // Schedule a periodic task with fixed delay:
    ScheduledExecutorService ses1 = Executors.newScheduledThreadPool(4);  // (3)
    try {
      System.out.println("Fixed delay:");
      ses1.scheduleWithFixedDelay(task, 1, 3, TimeUnit.SECONDS);          // (4)
      TimeUnit.SECONDS.sleep(15);                                         // (5)
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      ses1.shutdown();                                                    // (6)
    }

    // Schedule a periodic task at fixed rate:
    ScheduledExecutorService ses2 = Executors.newScheduledThreadPool(4);  // (7)
    try {
      System.out.println("\nFixed rate:");
      ses2.scheduleAtFixedRate(task, 1, 3, TimeUnit.SECONDS);             // (8)
      TimeUnit.SECONDS.sleep(10);                                         // (9)
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      ses2.shutdown();
    }
  }
}
```

```
    } finally {
      ses2.shutdown();
    }
  }
}
```

```
Fixed delay:
pool-1-thread-1: 12:55:54 Start:  I am on it!
pool-1-thread-1: 12:55:55 Finish: I am not on it!
pool-1-thread-1: 12:55:58 Start:  I am on it!
pool-1-thread-1: 12:55:59 Finish: I am not on it!
pool-1-thread-2: 12:56:02 Start:  I am on it!
pool-1-thread-2: 12:56:03 Finish: I am not on it!
pool-1-thread-1: 12:56:06 Start:  I am on it!
pool-1-thread-1: 12:56:07 Finish: I am not on it!

Fixed rate:
pool-1-thread-2: 12:56:09 Start:  I am on it!
pool-1-thread-2: 12:56:10 Finish: I am not on it!
pool-1-thread-4: 12:56:12 Start:  I am on it!
pool-1-thread-4: 12:56:13 Finish: I am not on it!
pool-1-thread-4: 12:56:15 Start:  I am on it!
pool-1-thread-4: 12:56:16 Finish: I am not on it!
pool-1-thread-4: 12:56:18 Start:  I am on it!
pool-1-thread-4: 12:56:19 Finish: I am not on it!
```

```
package executors;
import java.math.BigInteger;
import java.util.concurrent.*;

public class TaskCancellation {

  // Computation-intensive task:
  private static BigInteger bigFactorial(int number) {                  // (1)
    String threadName = Thread.currentThread().getName();
    BigInteger factorial = BigInteger.ONE;
    try {
      for (int i = 2; i <= number; i++) {
        factorial = factorial.multiply(BigInteger.valueOf(i));
        if (Thread.interrupted()) {                                      // (2)
          throw new InterruptedException();
        }
      }
      System.out.println(threadName + " completed.");
    } catch (InterruptedException e) {
      System.out.println(threadName + " interrupted.");                  // (3)
    }
    return factorial;
  }

  private static void timedTaskCancellation() {                         // (4)
    System.out.println("Timed task cancellation:");
    int number = 100000;
    long timeout = 1;
    TimeUnit unit = TimeUnit.SECONDS;
    System.out.println("NUMBER: " + number + ", "
                     + "TIMEOUT: " + timeout + " " + unit);
    ExecutorService exs = Executors.newSingleThreadExecutor();          // (5)
    Future<BigInteger> future = null;
    try {
      future = exs.submit(() -> bigFactorial(number));                  // (6)
      BigInteger result = future.get(timeout, unit);                    // (7)
      System.out.println("Factorial: " + result);                       // (8)
    } catch (TimeoutException e) {                                       // (9)
      System.out.println(e);
      System.out.println("cancel(true):  " + future.cancel(true));      // (10)
      System.out.println("isCancelled(): " + future.isCancelled());     // (11)
    } catch (InterruptedException | ExecutionException e) {
      System.out.println(e);                                             // (12)
    } finally {
      exs.shutdown();
    }
  }

  private static void scheduledTaskCancellation() {                     // (13)
    System.out.println("Scheduled task cancellation:");
    int number = 100000;
    long delay = 10;
    TimeUnit unit = TimeUnit.MILLISECONDS;
    System.out.println("NUMBER: " + number + ", DELAY: " + delay + " " + unit);
    ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);   // (14)
    try {
      Future<BigInteger> future = ses.submit(() -> bigFactorial(number)); // (15)
      ses.schedule(() -> future.cancel(true), delay, unit);               // (16)
      TimeUnit.SECONDS.sleep(1);                                          // (17)
      System.out.println("isCancelled(): " + future.isCancelled());       // (18)
    } catch (InterruptedException e) {
      System.out.println(e);
    } finally {
      ses.shutdown();
    }
  }

  public static void main(String[] args) {
    timedTaskCancellation();
    System.out.println();
    scheduledTaskCancellation();
  }
}
```

```
Timed task cancellation:
NUMBER: 100000, TIMEOUT: 1 SECONDS
java.util.concurrent.TimeoutException
pool-1-thread-1 interrupted.
cancel(true):  true
isCancelled(): true

Scheduled task cancellation:
NUMBER: 100000, DELAY: 10 MILLISECONDS
pool-2-thread-1 interrupted.
isCancelled(): true
```

```
if (taskSize < threshold)
  solve task directly                        // Base case
else {
  divide task into subtasks                  // Divide
  solve each subtask recursively             // Conquer
  combine results from each subtask          // Combine
}
```

```
package forkjoin;

import java.util.Random;
import java.util.concurrent.*;
import java.util.function.IntPredicate;
import java.util.stream.IntStream;

class FilterTask extends RecursiveTask<Integer> {                            // (1)
  public static final int THRESHOLD = 10_000;
  private int[] values;
  private int fromIndex;
  private int toIndex;
  private IntPredicate predicate;

  public FilterTask(int[] values, int fromIndex, int toIndex,
                    IntPredicate predicate) {
    this.values = values;
    this.fromIndex = fromIndex;
    this.toIndex = toIndex;
    this.predicate = predicate;
  }

  @Override
  protected Integer compute() {                                      // (2)
    if (toIndex - fromIndex < THRESHOLD) {
      // The base case:                                                 (3)
      var count = 0;
      for (int i = fromIndex; i < toIndex; i++) {
        if (predicate.test(values[i])) {
          ++count;
        }
      }
      return count;
    } else {
      // Create subtasks:                                               (4)
      var middel = fromIndex + (toIndex - fromIndex) / 2;
      var subtask1 = new FilterTask(values, fromIndex, middel, predicate);
      var subtask2 = new FilterTask(values, middel, toIndex, predicate);
      // Fork and execute the subtasks. Await completion.               (5)
      ForkJoinTask.invokeAll(subtask1, subtask2);
      // Combine the results returned by subtasks.                      (6)
      return subtask1.join() + subtask2.join();
    }
  }
}

public class ForkJoinDemo {
  public static void main(String[] args) {
    // Set up the array with the random int values:                     (7)
    final int SIZE = 1_000_000;
    int[] numbers = new Random().ints(SIZE).toArray();

    // Predicate to filter numbers divisible by 7:                      (8)
    IntPredicate predicate = i -> i % 7 == 0;

    // Create the initial task.                                         (9)
    var filterTask = new FilterTask(numbers, 0, numbers.length, predicate);
    // Create fork-join pool to manage execution of the task.           (10)
    var pool = new ForkJoinPool();
    // Perform the task, await completion, and return the result:       (11)
    var result = pool.invoke(filterTask);
    System.out.println("Fork/Join:       " + result);

    // Solution using parallel stream:                                  (12)
    System.out.println("Parallel stream: " +
                       IntStream.range(0, numbers.length).parallel()
                                .map(i -> numbers[i]).filter(predicate).count());
  }
}
```

```
package safe;
/** Interface that defines a basic counter. */
interface ICounter {                                                    // (1)
  void increment();
  int getValue();
}
```

```
package safe;
public class UnsafeCounter implements ICounter {                          // (2)
  private int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

```
package safe;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class TestCounters {                                          // (3)
  private static final int NUM_OF_INCREMENTS = 1000;
  private static final int POOL_SIZE = 10;

  public static void main(String[] args) throws InterruptedException {    // (4)

    UnsafeCounter usc = new UnsafeCounter();                         // (5)
    runIncrementor(usc);
    System.out.printf("Unsafe Counter: %24d%n", usc.getValue());

    VolatileCounter vc = new VolatileCounter();                      // (6)
    runIncrementor(vc);
    System.out.printf("Volatile Counter: %22d%n", vc.getValue());

    SynchronizedCounter sc = new SynchronizedCounter();              // (7)
    runIncrementor(sc);
    System.out.printf("Synchronized Counter: %18d%n", sc.getValue());

    AtomicCounter ac = new AtomicCounter();                          // (8)
    runIncrementor(ac);
    System.out.printf("Atomic Counter: %24d%n", ac.getValue());

    ReentrantLockCounter rlc = new ReentrantLockCounter();           // (9)
    runIncrementor(rlc);
    System.out.printf("Reentrant Lock Counter: %16d%n", rlc.getValue());

    ReentrantRWLockCounter rwlc = new ReentrantRWLockCounter();      // (10)
    runIncrementor(rwlc);
    System.out.printf("Reentrant Read-Write Lock Counter: %d%n", rwlc.getValue());
  }

  public static void runIncrementor(ICounter counter) {              // (11)
    // A Runnable incrementor to call the increment() method of the counter
    // a fixed number of times:
    Runnable incrementor = () -> {                                   // (12)
      IntStream.rangeClosed(1, NUM_OF_INCREMENTS).forEach(i->counter.increment());
    };

    // An executor service to manage a fixed number of incrementors:
    ExecutorService execService = Executors.newFixedThreadPool(POOL_SIZE);

    // Submit the incrementor to the executor service. Each thread executes
    // the same incrementor, and thereby increments the same counter.
    try {                                                            // (13)
      IntStream.range(0, POOL_SIZE).forEach(i -> execService.submit(incrementor));
    } finally {
      execService.shutdown();
    }
    // Wait for all tasks to complete.
    try {
      while (!execService.awaitTermination(1, TimeUnit.SECONDS));
    } catch (InterruptedException e) {
      execService.shutdownNow();
    }
  }
}
```

```
Unsafe Counter:                         8495
Volatile Counter:                       8765
Synchronized Counter:                  10000
Atomic Counter:                        10000
Reentrant Lock Counter:                10000
Reentrant Read-Write Lock Counter:     10000
```

```
package safe;
/** Potential problem with visibility of shared data. */
public class NonVolatileDemo {

  private static boolean stopThread = false;             // (1)

  public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 2; i++) {                        // (2)
      new Thread(() -> {
        while (!stopThread) {                            // (3)
          System.out.println(Thread.currentThread().getName()
                             + ": Get me out of here!");
        }
      }, "T" + i).start();
    }
    Thread.sleep(1);                                     // (4)
    stopThread = true;                                   // (5)
  }
}
```

```
package safe;
/** Volatile field to guarantee visibility of shared data. */
public class VolatileDemo {

  private static volatile boolean stopThread = false;    // (6)
```

```
    // The main() method remains the same as in the NonVolatileDemo class.

}
```

```
package safe;
public class VolatileCounter implements ICounter {
  private volatile int counter = 0;

  @Override public int getValue()   { return counter; }
  @Override public void increment() { ++counter; }
}
```

```
package safe;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter implements ICounter {
  private AtomicInteger counter = new AtomicInteger(0);                   // (1)

  @Override
  public int getValue() {                                                 // (2)
    return counter.get();
  }

  @Override
  public void increment() {                                               // (3)
    counter.incrementAndGet();                                            // (4)
//  while (true) {                                                        // (5)
//    int expectedValue = counter.get();                                  // (6)
//    int newValue = expectedValue + 1;                                   // (7)
//    if (counter.compareAndSet(expectedValue, newValue)) {// (8) Compare and Set.
//      return;                                            // (9)
//    }
//  }
  }
}
```

```
Atomic Counter:                      10000
```

```
AtomicInteger()
AtomicInteger(int initialValue)
```

```
boolean compareAndSet(int expectedValue, int newValue)
```

```
int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)
int updateAndGet(IntUnaryOperator updateFunction)
int getAndUpdate(IntUnaryOperator updateFunction)
```

```
package safe;
public class SynchronizedCounter implements ICounter  {
  private int counter = 0;

  @Override public synchronized int getValue() { return counter; }
  @Override public synchronized void increment() { counter++; }
}
```

```
Synchronized Counter:              10000
```

```
Lock frl = new ReentrantLock(true);  // (1) Fair reentrant lock
Lock rl = new ReentrantLock();       // (2) Reentrant lock with no fairness policy
```

```
frl.lock();                    // (1) Acquire the lock.
try {
  // Exclusive access...
} finally {
  frl.unlock();                // (2) Release the lock.
}
```

```
frl.lock();                                          // Unconditional locking.
try { /* Lock acquired. Can access the resource. */ }
finally { frl.unlock(); }
```

```
if (frl.tryLock()) {                                       // Polled locking.
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
} else { /* Lock was not acquired. */ }
```

```
try {
  if (frl.tryLock(100, TimeUnit.MILLISECONDS)) {           // Timed locking.
    try { /* Lock acquired. Can access the resource. */ }
    finally { frl.unlock(); }
  } else { /* Lock was not acquired. */ }
} catch (InterruptedException iex) { iex.printStackTrace(); }
```

```
try {
  frl.lockInterruptibly();                              // Interruptible locking.
  try { /* Lock acquired. Can access the resource. */ }
  finally { frl.unlock(); }
} catch (InterruptedException iex) { iex.printStackTrace(); }
```

```
boolean tryLock(long time, TimeUnit unit) throws InterruptedException
```

```
void lockInterruptibly() throws InterruptedException
```

```
boolean hasQueuedThread(Thread thread)
```

```
frl.lock();
frl.lock();
// Access the resource. Lock hold count is 2.
frl.unlock();
frl.unlock();
```

```
package safe;
import java.util.concurrent.locks.*;

public class ReentrantLockCounter implements ICounter {                    // (1)

  private Lock rl = new ReentrantLock();                                   // (2)
  private int counter = 0;                                                 // (3)

  @Override
  public int getValue() {                                                  // (4)
    rl.lock();
    try {
      return counter;
    } finally { rl.unlock(); }
  }

  @Override
  public void increment() {                                                // (5)
    rl.lock();
    try {
      counter++;                                                           // (6)
      getValue();                                                          // (7)
    } finally { rl.unlock(); }
  }
}
```

```
Reentrant Lock Counter:            10000
```

```
ReentrantReadWriteLock.ReadLock readLock()
ReentrantReadWriteLock.WriteLock writeLock()
```

```
ReentrantReadWriteLock(boolean fair)
```

```
package safe;
import java.util.OptionalInt;
import java.util.concurrent.locks.*;

public class ReentrantRWLockCounter implements ICounter {

  private ReadWriteLock rwl = new ReentrantReadWriteLock();           // (1)
  private Lock readLock = rwl.readLock();                             // (2)
  private Lock writeLock = rwl.writeLock();                           // (3)
  private int counter = 0;

  @Override
  public int getValue() {                                             // (4)
    readLock.lock();
    try {
//      System.out.println(Thread.currentThread().getName() + ": " + counter);
      return counter;
    } finally { readLock.unlock(); }
  }
  @Override
  public void increment() {                                        // (5)
    writeLock.lock();
    try {
      counter++;
    } finally { writeLock.unlock(); }
  }

  public int incrementAndGet() {                                   // (6)
    writeLock.lock();                // Acquire write lock.
    try {
      return ++counter;                                            // (7)
//    ++counter;                     // Increment.                 // (8)
//    return getValue();             // Get the new value.         // (9)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public int getAndIncrement() {                                   // (10)
    writeLock.lock();                // Acquire write lock.
    try {
      return counter++;              // Get and increment.         // (11)
    } finally { writeLock.unlock(); } // Release write lock.
  }

  public boolean incrIfPossible() {                                // (12)
    if (writeLock.tryLock()) {         // Attempts to acquire the write lock.
      try {                            // Write lock acquired.
        counter++;
        return true;
      } finally { writeLock.unlock(); } // Write lock released.
    } else {                           // Write lock not acquired.
      return false;
    }
  }

  public OptionalInt getIfPossible() {                             // (13)
    if (readLock.tryLock()) {          // Attempts to acquire the read lock.
      try { return OptionalInt.of(counter); }
      finally { readLock.unlock(); }   // Read lock released.
    } else {                           // Read lock not acquired.
      return OptionalInt.empty();
    }
  }
}
```

```
Reentrant Read-Write Lock Counter: 10000
```

```
boolean isWriteLockedByCurrentThread()
```

```
CyclicBarrier(int parties)
CyclicBarrier(int parties, Runnable barrierAction)
```

```
CyclicBarrier barrier1 = new CyclicBarrier(3);                      // (1)
CyclicBarrier barrier2 = new CyclicBarrier(5, () ->
        System.out.println("All results accumulated."));           // (2)
```

```
// Thread t1 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...

// Thread t2 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...

// Thread t3 waits on barrier1 at this barrier point.
barrier1.await();            // Can continue when the barrier is tripped.
...
```

```
int await() throws InterruptedException, BrokenBarrierException
int await(long timeout, TimeUnit unit) throws InterruptedException,
                                              BrokenBarrierException,
                                              TimeoutException
```

```
package synchronizers;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
  public static final int PARTIES = 3;

  public static void main(String args[]) {
    Runnable barrierAction =  () ->                                      // (1)
          System.out.println("Barrier action by "
                             + Thread.currentThread().getName()
                             + ": All tasks are released.");

    CyclicBarrier barrier = new CyclicBarrier(PARTIES, barrierAction);     // (2)

    Runnable task = () -> {                                              // (3)
      String threadName = Thread.currentThread().getName();
      try {
        System.out.println(threadName + " is now waiting");
        barrier.await();                                       // Barrier point.
        System.out.println(threadName + " is now released");
      } catch (BrokenBarrierException | InterruptedException e) {
        e.printStackTrace();
      }
    };

    for (int i = 0; i < PARTIES; i++) {                                  // (4)
      new Thread(task, "T" + (i+1)).start();
    }
  }
}
```

```
T1 is now waiting
T3 is now waiting
T2 is now waiting
Barrier action by T2: All tasks are released.
T2 is now released
T1 is now released
T3 is now released
```

```
void await() throws InterruptedException
boolean await(long timeout, TimeUnit unit) throws InterruptedException
```

```
CountDownLatch latch = new CountDownLatch(3);    // (1) Latch count is 3.
```

```
// Thread t1:
latch.await();            // (2) Waiting for latch to be raised.
...

// Thread t2:
latch.await();            // (3) Waiting for latch to be raised.
...
```

```
// Thread t3:
latch.countDown();
...
latch.countDown(),
...
latch.countDown();          // (4) Count reaches 0. Latch released at (2) and (3).
...
```

```
package synchronizers;
import static java.lang.System.out;
import java.util.concurrent.*;

public class CountDownLatchTest {
  public static final int N = 3;

  public static void main(String[] args) throws InterruptedException {
    CountDownLatch startLine = new CountDownLatch(1);              // (1)
    CountDownLatch finishLine = new CountDownLatch(N);            // (2)

    ExecutorService es = Executors.newFixedThreadPool(N);         // (3)
    String threadName = Thread.currentThread().getName();
    try {
      for (int i = 0; i < N; ++i) {    // (4) Submit tasks.
        es.submit(new Task(startLine, finishLine));
      }
      out.println(threadName + ": Let all tasks proceed.");
      startLine.countDown();           // (5) Count down to let all tasks proceed.
      finishLine.await();              // (6) Wait for all tasks to finish.
      out.println(threadName + ": All tasks done.");
    } finally {
      es.shutdown();
    }
  }
}

class Task implements Runnable {                                  // (7)
  private final CountDownLatch startLine;
  private final CountDownLatch finishLine;

  public Task(CountDownLatch start, CountDownLatch finish) {
    this.startLine = start;
    this.finishLine = finish;
  }

  @Override
  public void run() {                                              // (8)
    String threadName = Thread.currentThread().getName();
    try {
      out.println(threadName + ": Waiting to proceed.");
      startLine.await();               // (9) Wait to proceed.
      out.println(threadName + ": Running ... ");
      finishLine.countDown();          // (10) Count down the latch & continue.
      out.println(threadName + ": Latch count decremented.");
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

```
main: Let all tasks proceed.
pool-1-thread-3: Waiting to proceed.
pool-1-thread-2: Waiting to proceed.
pool-1-thread-1: Waiting to proceed.
pool-1-thread-1: Running ...
pool-1-thread-2: Running ...
pool-1-thread-2: Latch count decremented.
pool-1-thread-3: Running ...
pool-1-thread-1: Latch count decremented.
pool-1-thread-3: Latch count decremented.
main: All tasks done.
```

```
// Thread-safe unmodifiable list:
List<String> list = List.of("Tom", "Dick", "Harriet");
list.add("Harry");                          // UnsupportedOperationException
list.set(0, "Tommy");                       // UnsupportedOperationException
list.remove("Dick");                        // UnsupportedOperationException
System.out.println(list.get(0));            // "Tom"
```

```
// Effectively immutable list view:
List<Integer> list = new ArrayList<>();
list.add(2021); list.add(2022);
List<Integer> immutablelist = Collections.unmodifiableList(list);
immutablelist.add(2023);                     // UnsupportedOperationException
immutablelist.set(0, 2023);                  // UnsupportedOperationException
immutablelist.remove(2021);                  // UnsupportedOperationException
System.out.println(immutablelist.get(0));    // 2021
```

```
static <E> Collection<E>   synchronizedCollection(Collection<E> c)
static <E> List<E>         synchronizedList(List<E> list)
static <E> Set<E>          synchronizedSet(Set<E> set)
static <E> SortedSet<E>    synchronizedSortedSet(SortedSet<E> set)
static <E> NavigableSet<E> synchronizedNavigableSet(NavigableSet<E> set)
static <K,V> Map<K,V>       synchronizedMap(Map<K,V> map)
static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> map)
static <K,V> NavigableMap<K,V>
                            synchronizedNavigableMap(NavigableMap<K,V> map)
```

```
package synced;
import java.util.*;
import java.util.stream.IntStream;

public class SerialAccessThreads  {

  public static void main(String[] args) throws InterruptedException {
    List<Integer> years = Collections.synchronizedList(new ArrayList<>()); // (1)
    years.add(2024); years.add(2023); years.add(2021); years.add(2022);

    Runnable eliminator = () -> {                                         // (2)
      boolean found = false;
      synchronized(years) {                                               // (3)
        Iterator<Integer> iteratorA = years.iterator();                   // (4)
        while (iteratorA.hasNext()) {
          if (iteratorA.next().equals(2021)) {                            // (5)
```

```
            iteratorA.remove();                                          // (6)
            found = true;
          }
        }
      }                                                                  // (7)
      System.out.println("List modified: " + found);
    };
    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(eliminator).start());
  }
}
```

```
Exception in thread "Thread-1"
Exception in thread "Thread-2"
Exception in thread "Thread-0" java.lang.NullPointerException:
    Cannot invoke "java.lang.Integer.equals(Object)" because the return value of
    "java.util.Iterator.next()" is null
    ...
```

```
// Thread t1                                 // Thread t2
...                                          ...
if (name.size() > 0) // true                 ...
...                                          if (name.size() > 0) // true
return remove(0);    // Size 0               ...
...                                          return remove(0);    // Exception!
```

```
package synced;
import java.util.*;

public class DoubleAct {
  // Synchronized list:                                                       (1)
  private List<String> names = Collections.synchronizedList(new ArrayList<>());

  public void add(String name) { names.add(name); }                       // (2)

  public String removeFirst() {                                            // (3)
    synchronized(names) {                                                  // (4)
      if (names.size() > 0) {                                              // (5)
        try { Thread.sleep(1); }                                           // (6)
        catch(InterruptedException e) { e.printStackTrace(); }
        return names.remove(0);                                            // (7)
      } else { return null; }                                              // (8)
    }                                                                      // (9)
  }
}
```

```
package synced;
import java.util.stream.IntStream;

public class Client {                                                // (9)
  public static void main(String[] args) {
    DoubleAct da = new DoubleAct();
    da.add("Laurel"); da.add("Hardy");                              // (10)
    Runnable remover = () -> {                                      // (11)
      String name = da.removeFirst();                               // (12)
      System.out.println(name);
    };
```

```
package synced;
import java.util.stream.IntStream;

public class Client {                                                    // (9)
  public static void main(String[] args) {
    DoubleAct da = new DoubleAct();
    da.add("Laurel"); da.add("Hardy");                                   // (10)
    Runnable remover = () -> {                                           // (11)
      String name = da.removeFirst();                                    // (12)
      System.out.println(name);
    };

    IntStream.rangeClosed(1, 3).forEach(i -> new Thread(remover).start());// (13)
  }
}
```

```
Laurel
Hardy
Exception in thread "Thread-1" java.lang.IndexOutOfBoundsException:
      Index 0 out of bounds for length 0
      at ...
      at synced.DoubleAction.removeFirst(DoubleAction.java:15)
      at ...
```

```
// Range-view operations
NavigableSet<E> headSet(E toElement,   boolean inclusive)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

```
package concurrent;
import java.util.concurrent.TimeUnit;
public class ConcUtil {
  public static void snooze(int timeout, TimeUnit unit) {
    String threadName = Thread.currentThread().getName();
    try {
      unit.sleep(timeout);
    } catch (InterruptedException ex) {
      System.out.println(threadName + ": " + ex);
      Thread.currentThread().interrupt();         // Reinstate interrupt status.
    }
  }
}
```

```
public class ConcUtil {
  public static void snooze(int timeout, TimeUnit unit) {
    String threadName = Thread.currentThread().getName();
    try {
      unit.sleep(timeout);
    } catch (InterruptedException ex) {
      System.out.println(threadName + ": " + ex);
      Thread.currentThread().interrupt();         // Reinstate interrupt status.
    }
  }
}
```

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class ConcurrentSkipListSetDemo {

  private static ConcurrentSkipListSet<Integer> set;                    // (1)

  private static Runnable sumValues = () -> {                           // (2)
    String threadName = Thread.currentThread().getName();
    while (true) {
      int sum = 0;
      for (Integer v : set) {                                           // (3)
        sum += v;
      }
      System.out.printf(threadName + ": sum%9d%n", sum);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
      if (Thread.interrupted()) break;                                  // (4)
    }
  };

  private static Runnable removeValues = () -> {                        // (5)
    String threadName = Thread.currentThread().getName();
    while (true) {
      Integer value = set.pollFirst();
      if (value == null) continue;
      System.out.printf(threadName + ": removed%5d%n", value);
      ConcUtil.snooze(2, TimeUnit.SECONDS);
      if (Thread.interrupted()) break;                                  // (6)
    }
  };

  public static void main(String[] args) {
    // Create and populate the set:                                        (7)
    set = new ConcurrentSkipListSet<>();
    new Random().ints(10, 0, 1000).forEach(val -> set.add(val));
    System.out.println(set);

    // Create an executor service to execute two tasks:                    (8)
    ExecutorService exs = Executors.newFixedThreadPool(2);
    try {
      exs.submit(sumValues);
      exs.submit(removeValues);
```

```
        ConcUtil.snooze(5, TimeUnit.SECONDS);                                    // (9)
      } finally {
        System.out.println("Shutting down now.");
        exs.shutdownNow();                                                       // (10)
      }
    }
  }
```

```
[20, 100, 236, 263, 299, 359, 548, 552, 591, 686]
pool-1-thread-1: sum     3654
pool-1-thread-2: removed   20
pool-1-thread-1: sum     3634
pool-1-thread-2: removed  100
pool-1-thread-1: sum     3534
pool-1-thread-2: removed  236
Shutting down now.
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
```

```
interface ConcurrentMap<K,V> extends Map<K, V>  // §15.8, p. 831.

default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remapFunction)
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
default V computeIfPresent(K key,
               BiFunction<? super K,? super V,? extends V> remapFunction)

default void forEach(BiConsumer<? super K,? super V> action)

default V getOrDefault(Object key, V defaultValue)

default V merge(K key, V value,
                BiFunction<? super V,? super V,? extends V> remapFunction)

V putIfAbsent(K key, V value)
boolean remove(Object key, Object value)

V       replace(K key, V value)
boolean replace(K key, V oldValue, V newValue)
default void replaceAll(BiFunction<? super K,? super V,? extends V> func)
```

```
interface ConcurrentNavigableMap<K,V> extends ConcurrentMap<K,V>,
                                              NavigableMap<K,V>
NavigableSet<K> descendingKeySet()
ConcurrentNavigableMap<K,V> descendingMap()
```

```
ConcurrentNavigableMap<K,V> headMap(K toKey)
ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)
```

```
ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)
ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                   K toKey, boolean toInclusive)
```

```
ConcurrentNavigableMap<K,V> tailMap(K fromKey)
ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
```

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;
import java.util.function.Function;
import java.util.stream.Collectors;

public class ConcurrentHashMapDemo {

  private static ConcurrentHashMap<Integer, Long> map;
  public static final int NUM_OF_THROWS = 1000;

  private static Runnable diceResultsReader = () -> {                     // (1)
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcurrentHashMap.KeySetView<Integer, Long> keySetView = map.keySet();
      String output = "";
      for (Integer key : keySetView) {
        Long value = map.get(key);
        output += " " + "<" + key + "," + value + ">";
      }
      System.out.println(threadName + ": {" + output + " }");
      ConcUtil.snooze(1000, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
    }
  };
  private static Runnable diceResultRemover = () -> {                    // (2)
    String threadName = Thread.currentThread().getName();
    while (true) {
      ConcUtil.snooze(500, TimeUnit.MILLISECONDS);
      if (Thread.interrupted()) break;
      Integer key = ThreadLocalRandom.current().nextInt(1, 7);  // [1, 6]
      Long value = map.remove(key);
      if (value == null) continue;
      String removedEntry = threadName + ": removed "
                          + "<" + key + "," + value + ">";
      System.out.println(removedEntry);
    }
  };

  public static void main(String[] args) throws InterruptedException {
    map = new ConcurrentHashMap<>(6);                                     //  (3)
    new Random().ints(NUM_OF_THROWS, 1, 7)                                //  (4)
                .boxed()
                .parallel()
                .collect(Collectors.groupingByConcurrent(
                    Function.identity(),
                    () -> map,
                    Collectors.counting()));

    ExecutorService exs = Executors.newFixedThreadPool(3);
    try {
      exs.submit(diceResultsReader);                                      // (5)
      exs.submit(diceResultRemover);
      exs.submit(diceResultRemover);
      ConcUtil.snooze(5, TimeUnit.SECONDS);
    } finally {
      exs.shutdownNow();                                                  // (6)
    }
  }
}
```

```
pool-1-thread-1: { <1,160> <2,159> <3,170> <4,178> <5,158> <6,175> }
pool-1-thread-2: removed <1,160>
pool-1-thread-3: removed <6,175>
pool-1-thread-2: removed <5,158>
pool-1-thread-1: { <2,159> <3,170> <4,178> }
pool-1-thread-2: removed <4,178>
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-1: { <2,159> <3,170> }
pool-1-thread-2: removed <2,159>
pool-1-thread-1: { <3,170> }
pool-1-thread-2: removed <3,170>
pool-1-thread-1: java.lang.InterruptedException: sleep interrupted
pool-1-thread-2: java.lang.InterruptedException: sleep interrupted
pool-1-thread-3: java.lang.InterruptedException: sleep interrupted
```

```
void transfer(E e)
boolean tryTransfer(E e)
boolean tryTransfer(E e, long timeout, TimeUnit unit)
boolean hasWaitingConsumer()
int getWaitingConsumerCount()
```

```
package concurrent;
import java.util.concurrent.*;

public class LinkedBlockingQueueDemo {

  public static final int UPPER_BOUND = 3;
  public static final int NUM_OF_VALUES = 5;
  public static final int STOP_VALUE = -1;

  private static BlockingQueue<Integer> queue;                              // (1)

  private static Runnable producer = () -> {                                // (2)
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    try {
      for (int i = 0; i < NUM_OF_VALUES; i++) {
        Integer value = tlrng.nextInt(100);
        queue.put(value);                                                   // (3)
        System.out.println(threadName + ": put " + value);
        Thread.sleep(tlrng.nextInt(200));
      }
      queue.put(STOP_VALUE);                                                // (4)
      System.out.println(threadName + ": done.");
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    }
  };

  private static Runnable consumer = () -> {                                // (5)
    String threadName = Thread.currentThread().getName();
    ThreadLocalRandom tlrng = ThreadLocalRandom.current();
    while (true) {
      try {
        Integer head = queue.peek();                                        // (6)
        if (head != null && head.equals(STOP_VALUE)) {
            System.out.println(threadName + ": done.");
            break;
        }
        Integer value = queue.take();                                       // (7)
        System.out.println(threadName + ": processing " + value);
        Thread.sleep(tlrng.nextInt(1000));
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  };
  public static void main(String[] args) {
    queue = new LinkedBlockingQueue<>(UPPER_BOUND);                      // (8)
    ExecutorService exs = Executors.newFixedThreadPool(3);               // (9)
    try {
      exs.submit(producer);                                              // (10)
      exs.submit(consumer);
      exs.submit(consumer);
    } finally {
      exs.shutdown();                                                    // (11)
    }
  }
}
```

```
pool-1-thread-1: put 43
pool-1-thread-2: processing 43
pool-1-thread-1: put 84
pool-1-thread-3: processing 84
pool-1-thread-1: put 60
pool-1-thread-3: processing 60
pool-1-thread-1: put 55
pool-1-thread-1: put 17
pool-1-thread-3: processing 55
pool-1-thread-1: done.
pool-1-thread-3: processing 17
pool-1-thread-2: done.
pool-1-thread-3: done.
```

```
List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (1a)
cowlist.addAll(Arrays.asList(10, 20, 30));
Iterator<Integer> iter = cowlist.iterator();
while (iter.hasNext()) {
  Integer i = iter.next();
  if (i == 20) {
//  iter.remove();                       // (2a) UnsupportedOperationException
    cowlist.remove(i);                   // (2b) OK
    continue;
  }
  System.out.print(i + " ");             // With (2b): 10 30
}
```

```
    continue;
  }
  System.out.print(i + " ");              // With (2b): 10 30
}
```

```
int addAllAbsent(Collection<? extends E> c)
```

```
package concurrent;
import java.util.*;
import java.util.concurrent.*;

public class CopyOnWriteArrayListDemo {

  public static void main(String[] args) {
    List<Integer> cowlist = new CopyOnWriteArrayList<Integer>();          // (1)
    cowlist.addAll(Arrays.asList(1, 2, 3));

    Runnable iter = () -> {                                               // (2)
      String threadName = Thread.currentThread().getName();
      for (Integer i : cowlist) {
        System.out.println(threadName + ": " + i);
        ConcUtil.snooze(1, TimeUnit.SECONDS);
      }
    };
    // First iterator:
    new Thread(iter, "Iterator A").start();                              // (4)

    // Snooze, add, and remove in main thread.                           // (5)
    ConcUtil.snooze(1, TimeUnit.SECONDS);
    Integer newValue = 4;
    cowlist.add(newValue);
    System.out.println("New value added: " + newValue);
    Integer first = cowlist.remove(0);
    System.out.println("Value removed:   " + first);

    // Second iterator:
    new Thread(iter, "Iterator B").start();                              // (6)
  }
}
```

```
var values = List.of("1", "2", "3", "4", "5", "6", "7", "8");
var list1 = new CopyOnWriteArrayList<Integer>();
for (String value: values) {
    new Thread(() -> {
        list1.add(Integer.valueOf(value));
    }).start();
}
System.out.println(list1);

var list2 = values.parallelStream().map(v->Integer.valueOf(v))
                                   .toList();
System.out.println(list2);

var list3 = new ArrayList<Integer>();
values.parallelStream().map(v->Integer.valueOf(v)).forEach(v->list3.add(v));
System.out.println(list3);
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
public class RQ7 {
  public static void main(String[] args) {
    AtomicLong aValue = new AtomicLong(0);
    Callable<Number> c = () -> {
      AtomicLong value = aValue;
        return value.incrementAndGet();
      };

    Collection<Callable<Number>> cc = List.of(c, c, c);
    ExecutorService es = Executors.newFixedThreadPool(2);
    List<Future<Number>> futures = null;
    try {
      futures = es.invokeAll(cc);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    es.shutdown();

    futures.stream().mapToLong(v-> {
    try {
      return v.get().longValue();
    }catch(Exception e) {
      return -1;
    }}).forEach(v->System.out.print(v));
    System.out.print(aValue);
  }
}
```

```
import java.util.concurrent.locks.*;
public class RQ8 {
  public static void main(String[] args) {
    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    Lock rl = rwl.readLock();
    Lock wl = rwl.writeLock();
    try {
      rl.lock();
      System.out.println("Read lock acquired");
      if (wl.tryLock()) {
        System.out.println("Write lock acquired");
      }
    } catch(Exception e) {
      System.out.println("Lock acquisition failed");
    } finally {
      rl.unlock();
      if (rwl.isWriteLocked()) {
        wl.unlock();
        System.out.println("Write lock released");
      }
    }
    System.out.println("The end");
  }
}
```

```
public class RQ9 {
  private static volatile int counter = 10;
  public static void main(String[] args) {
    Runnable r = () -> counter--;
    while (counter > 0) {
      new Thread(r).start();
      System.out.print(counter);
    }
  }
}
```

```
import java.util.concurrent.*;
public class RQ10 {
  public static void main(String[] args) {
    try {
      ExecutorService es = Executors.newFixedThreadPool(2);
      Future<?> f1 = es.submit(() -> "acme");
      Future<?> f2 = es.submit(() -> {}); // (1)
      es.shutdown();                      // (2)
      Object o1 = f1.get();
      Object o2 = f2.get();               // (3)
      System.out.println(o1 + " " + o2);  // (4)
    } catch (InterruptedException | ExecutionException e) {
      e.printStackTrace();
    }
  }
}
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class RQ11 {
  public static void main(String[] args) {
    Map<Integer, String> map =
        new ConcurrentHashMap<>(Map.of(1,"a",2,"b",3,"c",4,"d",5,"e"));
    List<Future<String>> results = new CopyOnWriteArrayList<>();

    ExecutorService es = Executors.newFixedThreadPool(3);
    for (int i = 1; i <= map.size(); i++) {
      final int key = i;
      Future<String> f = es.submit(() -> map.get(key).toUpperCase());
      if (i % 2 != 0) {
        f.cancel(true);
      }
      results.add(f);
    }
    es.shutdown();
    String result = (results.stream().allMatch(r -> r.isDone()))
        ? results.stream().filter(r -> !r.isCancelled()).map(r -> {
            try {
              return r.get();
            } catch (InterruptedException | ExecutionException e) {
              return "X";
            }
          }).collect(Collectors.joining())
        : "Z";
    System.out.println(result);
  }
}
```

```
CREATE TABLE table_name (column_definitions);
```

```
column_name column_type constraints, ...
```

```
CREATE TABLE compositions (isrc VARCHAR(12) PRIMARY KEY,
                           title VARCHAR(40) NOT NULL, duration INT NOT NULL);
```

```
INSERT INTO table_name VALUES (actual_values);
```

```sql
INSERT INTO compositions VALUES ('ushm91736697', 'Vacation', 231);
```

```
SELECT column_list FROM table_name WHERE row_filter;
```

```
column_name = some_value AND column_name LIKE some_value
```

```sql
SELECT isrc, title FROM compositions WHERE duration > 200 AND title LIKE 'V%';
```

```
UPDATE table_name SET column_name_value_pairs WHERE row_filter;
```

```
column_name = column_value, ...
```

```
UPDATE compositions SET duration = 240 WHERE isrc LIKE '%91736697'
                                                  OR duration > 231;
```

```
UPDATE compositions SET duration = 250;
```

```
DELETE FROM table_name WHERE row_filter;
DELETE FROM compositions WHERE isrc = 'ushm91736697';
```

```
try {
/* execute JDBC operations */
} catch (SQLException e) {
  String state = e.getSQLState();
  int code = e.getErrorCode();
}
```

```
try {
  /* establish database connection     */
  /* create and execute SQL statements */
  /* process results                   */
} catch (SQLException e) {
  /* handle any errors */
} finally {
  /* close result sets */
  /* close statements  */
```

```
    /* close connection  */
  }
```

```
try (/* establish database connection     */
     /* create and execute SQL statements */)
{
  /* process results */
} catch (SQLException e) {
  /* handle any exceptions */
} /* implicit finally block closes resources*/
```

```
static Connection getConnection(String jdbcURL)
static Connection getConnection(String jdbcURL, String username,
                                String password)
static Connection getConnection(String jdbcURL, Properties info)
```

protocol:*provider*:*driver_type*:*database_specific_connection_details*

```
jdbc:derby:localhost:1521:musicDB
```

```
jdbc:oracle:thin:@localhost:1521:musicDB
```

```
String jdbcURL  = "jdbc:derby:localhost:1521:musicDB";
String username = "joe";
String password = "welcome1";
try (Connection connection = DriverManager.getConnection(jdbcURL,
                                                         username, password)) {
  /* use the connection. */
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
Working directory
 ?
 ??? db-derby-10.15.2.0-lib/  <== Directory with the Apache Derby distribution
 ?   ?
 ?   ??? lib/                 <== Directory with the Derby JAR files
 ?
 ??? musicDB/                 <== Directory with the musicDB database files
 ?
 ??? dbDemo/                  <== Package with examples used in this chapter
 ?   ?
 ?   ??? JDBCConnection.java
 ?   ?
... ...
```

```
package dbDemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCConnection {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (Connection connection = DriverManager.getConnection(jdbcURL)) {
      /* use the connection. */
      System.out.println(connection);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
org.apache.derby.impl.jdbc.EmbedConnection@1201454821 (XID = 1301),
(SESSIONID = 1), (DATABASE = musicDB), (DRDAID = null)
```

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotfoundException c) { }
```

```
java -Djdbc.drivers=oracle.jdbc.driver.OracleDriver ...
```

```
Statement         createStatement()
PreparedStatement prepareStatement(String sql)
CallableStatement prepareCall(String sql)
```

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) { // Obtain a Statement object.
  /* execute a query */
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
boolean execute(String sql)
```

`ResultSet executeQuery(String sql)`

```
int executeUpdate(String sql)
```

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {      // Obtain a Statement object.
  String sql = "select * from compositions";    // SELECT query: select all rows.
  boolean isSelectStmt = statement.execute(sql);          // (1) Execute the query
  System.out.println("SELECT statement? " + isSelectStmt);// (2) SELECT statement?
                                                          //     true
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
boolean isSelectStmt = statement.execute(sql);
if (isSelectStmt) {
  try (ResultSet resultSet
          = statement.getResultSet()) { // SELECT statement: retrieve ResultSet.
    System.out.println("SELECT statement: processing ResultSet");
  }
} else {                                               // Update statement:
  int rowCount = statement.getUpdateCount();           // Retrieve the number of
                                                       // rows affected.
  System.out.println("Update statement: Rows affected " + rowCount);
}
```

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {          // Obtain a Statement object.
  String sql = "select * from compositions";        // SQL query: select all rows.
  try (ResultSet resultSet
           = statement.executeQuery(sql)) {    // (1) Nested try-with-resources
   System.out.println("Processing ResultSet");
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
try (Connection connection = DriverManager.getConnection(jdbcURL);      // (1)
     Statement statement = connection.createStatement()) {              // (2)
  String sql = "update compositions set duration = duration * 2";       // (3)
  int count = statement.executeUpdate(sql);                             // (4)
  System.out.println("Rows modified: " + count);
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
ResultSet resultSet = statement.executeQuery(
    "select * from compositions where title like '" + value + "'");
```

```
String value = "SomeValue'; drop table compositions;";
```

```
ResultSet resultSet2 = statement.executeQuery(
        "select * from compositions where title like '" +
        "SomeValue'; drop table compositions;");
```

```
String sql = "select * from compositions where duration > 200"; // SQL operation
try (Connection connection
         = DriverManager.getConnection(jdbcURL);                // Get connection
     PreparedStatement pStatement
         = connection.prepareStatement(sql)) {            // (1) Prepare statement
  boolean result = pStatement.execute();                  // (2) Execute
  System.out.println(result);
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
String sql = "select * from compositions where duration > ?"; // (1) SQL operation
                                                       // with 1 marker parameter.
PreparedStatement pStatement
    = connection.prepareStatement(sql);                // (2) A prepared statement
```

```
pStatement.setInt(1,200);           // Sets the value of the marker parameter
                                    // at position 1 to 200.
```

```
pStatement.setObject(1, 200, Types.INTEGER);
```

```
void setString(int index, String value)
void setBoolean(int index, boolean value)
void setInt(int index, int value)
void setLong(int index, long value)
void setDouble(int index, double value)
void setBigDecimal(int index, BigDecimal value)
void setDate(int index, Date value)
void setObject(int index, Object value, int sqlType)
```

```
package dbDemo;
import java.sql.*;

public class XQTPreparedStatement {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    String sql = "select * from compositions where duration > ?";           // (1)
    try (Connection connection = DriverManager.getConnection(jdbcURL);      // (2)
        PreparedStatement pStatement = connection.prepareStatement(sql);) { // (3)
      pStatement.setInt(1, 200);                                            // (4)
      boolean result = pStatement.execute();                                // (5)
      System.out.println(result);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
pStatement.setInt(1, 100);
result = pStatement.execute();
```

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;

public class PreparedStatementExecuteUpdate {
  public static void main(String[] args) {

    final String insSql = "insert into compositions VALUES(?, ?, ?)";
    final String updSql = "update compositions set title = ? where title = ?";
    final String delSql = "delete from compositions where duration = ?";

    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var pStatement1 = connection.prepareStatement(insSql);
        var pStatement2 = connection.prepareStatement(updSql);
        var pStatement3 = connection.prepareStatement(delSql)) {

        pStatement1.setInt(3, 150);
        pStatement1.setString(2, "Java Jazz");
        pStatement1.setString(1, "ushm91736991");
        int result1 = pStatement1.executeUpdate();
        System.out.println(result1);

        pStatement2.setString(1, "Java Jive");
        pStatement2.setString(2, "Java Jazz");
        int result2 = pStatement2.executeUpdate();
        System.out.println(result2);

        pStatement3.setInt(1, 200);
        int result3 = pStatement3.executeUpdate();
        System.out.println(result3);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
String procedureCall
    = "{ call some_procedure(?,?) }";          // (1) Call a stored procedure.
String functionCall = "? = { call some_function(?) }"; // (2) Call a function.
CallableStatement cStatement1 = connection.prepareCall(procedureCall); // (3)
CallableStatement cStatement2 = connection.prepareCall(functionCall);  // (4)
```

```
cStatement1.setString(1, "Hi");               // Set value of IN parameter
                                              // at marker 1
cStatement2.setObject(2, 42, Types.INTEGER);  // Set value of IN parameter
                                              // at marker 2
```

```
cStatement1.registerOutParameter(2, Types.VARCHAR); // Register SQL VARCHAR as
                                                     // type for the
                                                     // 2nd marker parameter.
cStatement2.registerOutParameter(1, Types.INTEGER); // Register SQL INTEGER as
                                                     // type for the
                                                     // 1st marker parameter.
```

```
void registerOutParameter(int parameterIndex, int sqlType)
```

```
boolean result1 = cStatement1.execute();
boolean result2 = cStatement2.execute();
```

```
String result1 = cStatement1.getString(2);     // Retrieve value of OUT parameter
                                               // at marker 2.
```

```
String result2 = cStatement2.getObject(1, Integer.class);// Retrieve value of OUT
                                                        // parameter at marker 1.
```

```
String getString(int paramIndex)
```

```
int getBoolean(int paramIndex)
int getInt(int paramIndex)
```

int getInt(int paramIndex)

```
double getLong(int paramIndex)
double getDouble(int paramIndex)
```

```
BigDecimal getBigDecimal(int paramIndex)
```

```
<T> T getObject(int paramIndex, Class<T> type)
```

```
public void storedProcedureCall(Connection connection) {
  final String callProc
     = "{call longCompositionsProc(?, ?)}";            // (1) 1: IN 2:OUT
  try (CallableStatement cStmt = connection.prepareCall(callProc)) { // (2)
    int duration = 100;                                              // (3)
    cStmt.setInt(1, duration);                                       // (4)
    cStmt.registerOutParameter(2, Types.INTEGER);                    // (5)
    cStmt.execute();                                                 // (6)
    int returnedValue = cStmt.getInt(2);                             // (7)
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {
    e.printStackTrace();
  }
}

public void storedFunctionCall(Connection connection) {
  final String callFunc
     = "? = {call longCompositionsFunc(?)}";            // (8) 1:OUT 2:IN
  try (CallableStatement cStmt = connection.prepareCall(callFunc)) {
    cStmt.registerOutParameter(1, Types.INTEGER);                     // (9)
    int duration = 100;
    cStmt.setInt(2, duration);                                        // (10)
    cStmt.execute();
    int returnedValue = cStmt.getInt(1);                              // (11)
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
boolean absolute(int rowNumber)
```

```
PreparedStatement statement
    = connection.prepareStatement("select * from compositions");
statement.setFetchSize(20);            // Rows would be downloaded 20 at a time.
ResultSet resultSet = statement.executeQuery();
while (resultSet.next()){
 /* Rows are processed one at a time regardless of the fetch size */
  if (/* some condition */) {
    resultSet.setFetchSize(10);        // Rows would be downloaded 10 at a time.
  }
}
```

```
String strValue1 = resultSet.getString(1);                // Using column index.
String strValue2 = resultSet.getString("column_name");    // Using column name.
String strValue3 = resultSet.getObject(1, String.class);
String strValue4 = resultSet.getObject("column_name", String.class);
String getString(int columnIndex)
String getString(String columnLabel)
```

```
int getBoolean(int columnIndex)
int getBoolean(String columnLabel)
int getInt(int columnIndex)
int getInt(String columnLabel)
```

```
double getLong(int columnIndex)
double getLong(String columnLabel)
double getDouble(int columnIndex)
double getDouble(String columnLabel)
```

```
BigDecimal getBigDecimal(int columnIndex)
BigDecimal getBigDecimal(String columnLabel)
```

```
Date getDate(int columnIndex)
Date getDate(int columnLabel)
```

```
<T> T getObject(int columnIndex, Class<T> type)
<T> T getObject(int columnLabel, Class<T> type)
```

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.time.Duration;

import static java.lang.System.out;
```

```
public class ResultSetProcessing {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    final String sql = "select * from compositions where duration > ?";    // (1)
    try (var connection = DriverManager.getConnection(jdbcURL);            // (2)
        var pStatement = connection.prepareStatement(sql);) {              // (3)
      pStatement.setInt(1, 0);                                             // (4)
      var resultSet = pStatement.executeQuery();
      try (resultSet) {                                                    // (5)
        while (resultSet.next()) {                                         // (6)
          String isrc = resultSet.getString(1);                            // (7)
          String title = resultSet.getObject(2, String.class);             // (8)
          int duration = resultSet.getInt("duration");                     // (9)
          out.println("[" + isrc + ", " + title + ", " + duration + "]"); // (10)
        }
      } // Closes the result set.
    } catch (SQLException e) {                                             // (11)
      e.printStackTrace();
    } // Closes the prepared statement and the connection.
  }
}
```

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
[ushm91736699, Why Don't, 178]
```

```
PreparedStatement prepareStatement(String sql,
                               int resultSetType, int resultSetConcurrency,
                               int resultSetHoldability)
```

```
package dbDemo;
import java.sql.DriverManager;

import java.sql.ResultSet;
import java.sql.SQLException;
public class ResultSetCustomization {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var statement = connection.prepareStatement(
            "select duration from compositions where title = ?",
            ResultSet.TYPE_FORWARD_ONLY,        // Forward direction. May reflect
                                                // database changes.
            ResultSet.CONCUR_UPDATABLE,         // Result set is updatable.
            ResultSet.CLOSE_CURSORS_AT_COMMIT   // Result set is closed on commit.
            )) {
      connection.setAutoCommit(false);          // (1) Disables automatic commit.
      statement.setString(1,"Vacation");
      try (ResultSet resultSet = statement.executeQuery();) {
        if (resultSet.next()) {                 // Moves forward one row.
          resultSet.updateInt("duration", 147); // Updates the current row
                                                // in the result set.
          resultSet.updateRow();                // Updates the underlying
                                                // database.
          System.out.println("Updated");
        }
        connection.commit();                    // (2) Also closes the result set.
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
String getDatabaseProductName()
```

`String getDatabaseProductVersion()`

```
boolean supportsResultSetType(int resultSettype)
boolean supportsResultSetHoldability(int holdability)
boolean supportsResultSetConcurrency(int resultSettype, int concurrency)
```

```
String getColumnName(int column)
```

```
package dbDemo;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DBMetadata {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // Print various information about the database:
      DatabaseMetaData dbMetaData
          = connection.getMetaData();                 // Obtain DatabaseMetaData.
      String dbName              = dbMetaData.getDatabaseProductName();
      String dbVersion           = dbMetaData.getDatabaseProductVersion();
      String sqlKeywords         = dbMetaData.getSQLKeywords();
      boolean forwardOnly = dbMetaData.supportsResultSetType(
          ResultSet.TYPE_FORWARD_ONLY);
      boolean cursorOpen  = dbMetaData.supportsResultSetHoldability(
          ResultSet.HOLD_CURSORS_OVER_COMMIT);
      boolean forwardUpdate = dbMetaData.supportsResultSetConcurrency(
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_UPDATABLE);

      System.out.println("Various info about the database:");
      System.out.println("Database name: " + dbName);
      System.out.println("Version: " + dbVersion);
      System.out.println("SQL keywords: " + sqlKeywords);
      System.out.println("TYPE_FORWARD_ONLY: " + forwardOnly);
      System.out.println("HOLD_CURSORS_OVER_COMMIT: " + cursorOpen);
      System.out.println("TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: "
          + forwardUpdate);

      // Create a ResultSet and print its structure:
      String sql = "select * from compositions where duration > ?";
      try (var pStatement = connection.prepareStatement(sql);) {
        pStatement.setInt(1, 100);
        var resultSet = pStatement.executeQuery();
        try (resultSet){
          System.out.println("Structure of ResultSet:");
          ResultSetMetaData rsMetaData
              = resultSet.getMetaData();              // Obtain ResultSetMetadata.
          int columnCount = rsMetaData.getColumnCount();
          System.out.println("Number of columns:" + columnCount);
          for (int i = 1; i <= columnCount; i++){
            String name = rsMetaData.getColumnName(i);
            int type = rsMetaData.getColumnType(i);   // Value of Types constant.
            System.out.println(name + ": " + type);
          }
        }
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
Various info about the database:
Database name: Apache Derby
Version: 10.15.2.0 - (1873585)
SQL keywords: ALIAS,BIGINT,BOOLEAN,CALL,CLASS,COPY,DB2J_DEBUG,EXECUTE,… (EDITED)
TYPE_FORWARD_ONLY: true
HOLD_CURSORS_OVER_COMMIT: true
TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: true
Structure of ResultSet:
Number of columns:3
ISRC: 12
TITLE: 12
DURATION: 4
```

```
connection.setAutoCommit(false);
```

```
void setAutoCommit(boolean autoCommit)
```

```
Savepoint setSavepoint()
Savepoint setSavepoint(String name)
```

```
void rollback()
void rollback(Savepoint savepoint)
```

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Savepoint;

public class Transactions {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // SQL statements:
      final String insSql = "insert into compositions VALUES(?, ?, ?)";
      final String updSql = "update compositions set title = ? where title = ?";
      final String delSql = "delete from compositions where duration = ?";
      // Create statements:
      try (var insStatement = connection.prepareStatement(insSql);
          var updStatement = connection.prepareStatement(updSql);
          var delStatement = connection.prepareStatement(delSql);) {

        connection.setAutoCommit(false);            // (1) Auto-commit disabled.
        insStatement.setInt(3, 150);                // (2) Insert a new row.
        insStatement.setString(2, "Java Jazz");
        insStatement.setString(1, "ushm91736991");
        int insResult = insStatement.executeUpdate();
        System.out.println("INSERT: " + insResult);

        updStatement.setString(1, "Java Jive");     // (3) Update an existing row.
        updStatement.setString(2, "Rage");
        int updResult = updStatement.executeUpdate();
        System.out.println("UPDATE: " + updResult);

        Savepoint savePoint = connection.setSavepoint(); // (4) Set a savepoint.
        delStatement.setInt(1, 178);                     // (5) Delete a row.
        int delResult = delStatement.executeUpdate();
        System.out.println("DELETE: " + delResult);

        connection.rollback(savePoint);   // (6) Roll back to safepoint.
        connection.commit();              // (7) Commits only (2) and (3).
      } catch (SQLException e) {
        connection.rollback();            // (8) Roll back any changes.
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
[ushm91736699, Why Don't, 178]
```

```
[ushm91736991, Java Jazz, 150}
```

[ushm91736698, Java Jive, 308]

[ushm91736699, Why Don't, 178]

```
[ushm91736697, Vacation, 231]
[ushm91736698, Java Jive, 308]
[ushm91736699, Why Don't, 178]
[ushm91736991, Java Jazz, 150]
```

```
String findAnswers = "select * from questions where question like ?";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findAnswers)) {
  ps.setString(1, "Where%");
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      String question = rs.getObject(2, String.class);   // (1)
      String answer = rs.getObject(3, String.class);     // (2)
      answer = (answer == null) ? "No answer." : answer; // (3)
      System.out.println(question + " - " + answer);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

```
String findQuestion = "select * from questions where id = ?";
int id = 103;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findQuestion)) {
  ps.setInt(1, id);                                  // (1)
  try (ResultSet rs = ps.executeQuery()) {
    if (rs.next()) {                                 // (2)
      String question = rs.getObject(2, String.class);
      System.out.println(question);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

```
String findQuestion = "select question from questions where id = ?";
String provideAnswer = "update questions set answer = ? where id = ?";
int id = 102;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps1 = c.prepareStatement(findQuestion);
     PreparedStatement ps2 = c.prepareStatement(provideAnswer)) {
  c.setAutoCommit(false);
  ps2.setString(1, "Look and you will find!");
  ps2.setInt(2, id);
  ps2.executeUpdate();
  ps1.setInt(2, id);
  try (ResultSet rs = ps1.executeQuery()) {
    while (rs.next()) {
      String question = rs.getString(1);
      String answer = rs.getString(2);
      System.out.println(question + " - " + answer);
    }
  }
```

```
    c.commit();
} catch (SQLException e) {
    e.printStackTrace();
}
```

```
try (Connection connection
         = DriverManager.getConnection(jdbcUrl, username, password);
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(sqlQuery)) {
  while (resultSet.next()) {
    /* Do nothing. */
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
String findQuestion
    = "select question, answer from questions where answer is null";
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findQuestion,
         ResultSet.TYPE_FORWARD_ONLY,
         ResultSet.CONCUR_UPDATABLE,
         ResultSet.CLOSE_CURSORS_AT_COMMIT)) {
  c.setAutoCommit(false);
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      rs.updateString("answer", "no answer");
      rs.updateRow();
    }
  }
  c.commit();
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
    var ps = c.prepareStatement(
                   "update questions set answer = ? where question = ?")) {
  ps.setString(2, "What?");                  // (1)
  try (ResultSet rs = ps.executeQuery()) {  // (2)
    while (rs.next()) {                      // (3)
      rs.setString(1, "42");                 // (4)
    }
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
    PreparedStatement ps = c.prepareStatement(
        "update questions set answer = ? where question = ?")) {
  ps.setString(2,"What?");          // (1)
  ps.setString(1,"42");             // (2)
  ps.execute();                     // (3)
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
import java.lang.annotation.*;
```

```
// Annotation type declaration with the name Tag.
@interface Tag {}              // Declaration always specified with
                               // at-sign (@) preceding the keyword interface.
```

```
// Applying annotation @Tag which denotes the annotation type declaration Tag.
@Tag class Gizmo {             // Always applied with at-sign (@) preceding
                               // the annotation type name.
  @Tag void start() {}         // Another invocation of the annotation type Tag.
}
```

```
meta-annotations access_modifier @interface annotation_name // Annot type header
{ // Annotation type body with zero or more member declarations:

  annotation_type_element_declarations

  constant_variable_declarations

  enum_declarations
  class_declarations
  interface_declarations
}
```

```
import java.lang.annotation.Target;                      // (1)
import java.lang.annotation.ElementType;                 // (2)
import java.lang.annotation.Retention;                   // (3)
import java.lang.annotation.RetentionPolicy;             // (4)

@Target({ElementType.TYPE, ElementType.METHOD})          // (5) Meta-annotation
@Retention(RetentionPolicy.RUNTIME)                      // (6) Meta-annotation
```

```
public @interface TaskInfo {                           // (7)
  String taskDesc();                                   // (8) Annotation type element
  String[] assignedTo();                               // (9) Annotation type element
  TaskPriority priority() default TaskPriority.NORMAL; // (10) Annot type element

  public enum TaskPriority { LOW, NORMAL, HIGH };      // (11) Nested enum type

  public static final String LOG_FILE = "./logs/Tasks.log"; // (12) const decl.
}
```

```
element_modifiers element_type method_name() default constant_expression;
```

```
// ...
public @interface TaskInfo {                        // (7)
  String taskDesc();                                // (8) Required annotation element
  String[] assignedTo();                            // (9) Required annotation element
  TaskPriority priority() default TaskPriority.NORMAL;  // (10) Optional
                                                        //      annotation element
  // ...
}
```

```
@interface Tag {}                   // No element type declarations.
```

```
@interface SecurityLevel {          // Single-element type declaration
  int value();
}
```

```
public @interface MultiElementAnnotationType {

  public enum Priority { LOW, NORMAL, HIGH };

  public int certificationLevel()     default 1;               // int
  String date()                       default "2021-01-11";    // String
  Class<? extends PrettyPrinter> pp()
                       default AdvancedPrettyPrinter.class;    // type Class
  Priority priorityLevel()            default Priority.NORMAL; // enum Priority
  Tag annotate()                      default @Tag;            // Annotation type
  int[] value()                       default {10, 20, 30};    // Array, int[]
}
```

```
// Auxiliary classes:
class PrettyPrinter {}
class AdvancedPrettyPrinter extends PrettyPrinter {}
```

```
@interface ProblematicAnnotationType {
  StringBuilder message();             // Illegal return type.
  int[][] voting();                    // Only one-dimensional array allowed.
  String value;                        // Missing parentheses.
  private Thread.State state();        // Only public can be specified.
}
```

```
public @interface MusicMeta {           // (1) Contained annotation type
  String value();
}

public @interface ArtistMeta {          // (2) Containing annotation type
  MusicMeta value();                    // Annotation type from (1).
}
```

```
@interface Refactor {
  int id()          default (int) (Math.random() * 10);// Not a const expression.
  String value()    default null;                      // Cannot be null.
  String deadline() default new String("2021-01-11");  // Not a const expression.
  String[] team()   default new String[] {"VJ", "PT"}; // Not a const expression.
}
```

```
// ...
public @interface TaskInfo {                        // (7)
  String taskDesc();                                // (8) Required annotation element
  String[] assignedTo();                            // (9) Required annotation element
  TaskPriority priority() default TaskPriority.NORMAL; // (10) Optional
                                                       //      annotation element
  // ...
}
```

```
@TaskInfo(                                                  // Normal annotation
   taskDesc = "Class for monitoring nuclear reactor activity",       // Required
   assignedTo = {"Tom", "Dick", "Harriet"},                          // Required
   priority = TaskInfo.TaskPriority.HIGH                             // Optional
   )
class NuclearPlant {}
```

```
@TaskInfo(                                                              // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",       // Required
   assignedTo = {"Tom", "Dick", "Harriet"}                            // Required
   )
class NuclearPlant {}
```

```
@TaskInfo(                                                     // Normal
annotation
   taskDesc = "Class for monitoring nuclear reactor activity",       // Required
   assignedTo = {"Tom", "Dick", "Harriet"},                          // Required
   priority = TaskInfo.TaskPriority.NORMAL                           // Implied
   )
class NuclearPlant {}
```

```
@TaskInfo(
    priority = TaskInfo.TaskPriority.LOW,
    taskDesc = "Start nuclear reactor",
    assignedTo = "Harriet"                // Single-element array-valued element
    )
class NuclearPlant {}
```

```
// Marker annotation type
@interface Tag {}                        // No element type declarations
```

```
@Tag class Gizmo {}                         // Marker annotation
```

```
@Tag() class Gizmo {}                       // Normal annotation
```

```
// Annotation type declaration where all elements have default values.
@interface Option {
  Color color()  default Color.WHITE;   // Optional annotation element
  Size size()    default Size.M;        // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

```
@Option                                              // Marker annotation
class Item {}
```

```
@Option(color = Option.Color.WHITE, size = Option.Size.M)  // Normal annotation
class Item {}
```

```
// Single-element annotation type
@interface Author {
  String value();                              // Single annotation type element
}
```

```
@Author("Tom")                                  // Single-element annotation
class Connection {}
```

```
@Author(value = "Tom")                          // Normal annotation
class Connection {}
```

```
//Annotation type declaration with a value() element and other elements with
default values.
@interface ExtraOption {
  int value();                               // Required annotation element
  Color color()  default Color.WHITE;        // Optional annotation element
  Size size()    default Size.M;             // Optional annotation element
  enum Color {RED, WHITE, BLUE}
  enum Size  {S, M, L, XL}
}
```

```
@ExtraOption(10)                                    // Single-element annotation
class ItemV2 {}
```

```
@ExtraOption(value=10, color=ExtraOption.Color.WHITE,      // Normal annotation
             size=ExtraOption.Size.M)
class ItemV3 {}
```

```
// Array-valued single-element annotation type declaration:
@interface Problems {
  String[] value();                                 // Array-type value() element
}

@Problems({"Code smell", "Exception not caught"})  // Array-valued single-element
class ItemV4 {}                                     // annotation

@Problems(value = {"Code smell", "Exception not caught"})  // Normal annotation
class ItemV5 {}
```

```
@Problems("Code smell")   // Single-element array-valued single-element annotation
class ItemV6 {}

@Problems({"Code smell"}) // Single-element array-valued single-element annotation
class ItemV7 {}

@Problems(value = "Code smell")      // Normal annotation
class ItemV8 {}

@Problems(value = {"Code smell"})    // Normal annotation
class ItemV9 {}
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value = RetentionPolicy.RUNTIME)   // Retention: In source code,
                                              // in class file, at runtime.
public @interface MusicMeta {}
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value = RetentionPolicy.SOURCE) // Retention: Only in the source code.
public @interface Override {}
```

```
@Tag public static final int YEAR = 2021;
enum Size { @Tag S, @Tag M, @Tag L, @Tag XL }
```

```
@Tag String localBarber = "Director's Cut";
for (@Tag int i : intArray) {}
try (@Tag var resource = new CloseableResource()) {}
```

```
@Tag boolean passed() { return true; }
@Tag String[] value();
```

```
@Tag package com.passion.logic;
```

```
double circleArea(@Tag double r) { return Math.PI*r*r; }
Predicate<String> p = (@Tag var str) -> str.length() < 10;
try {} catch (@Tag Exception ex) {}
```

```
@Tag class Gizmo {}
@Tag interface Reparable {}
@Tag enum Direction {LEFT, RIGHT}
@Tag @interface Validate {}
```

```
class Box<@Tag E> {}
interface Eatable<@Tag E> {}
<@Tag E extends Comparable<E>> void comparison(E e) {}
<@Tag E> Item(E e) {}
```

```
class X extends @Tag Y implements @Tag IZ {}
java.lang. @Tag String strOp() { return "ok"; }
@Tag int compute() { return (@Tag int) 3.14; }
java.lang. @Tag Integer iRef1 = 100;
@Tag Integer iRef2 = 100;
void passItOn() throws @Tag Exception {}
@Tag int[] array1;          // Annotates primitive type int
int @Tag [] array2;         // Annotates array type int[]
int @Tag [][] array3;       // Annotates array type int[][]
```

```
@Target(ElementType.CONSTRUCTOR)
@interface Tag {}
______________________________
class Item {
  @Tag public Item() {}       // Declaration context. Annotates constructor Item.
}
```

```
@Target({ElementType.CONSTRUCTOR, ElementType.PARAMETER})
@interface Tag {}
-------------------------------
double circleArea(@Tag double r) { // Declaration context. Annotates parameter r.
  return 5.0;
}
```

```
@Target(ElementType.TYPE_USE)
@interface Tag {}
_______________________________
java.lang. @Tag Integer iRef1 = 100;        // (1) Type context. Annotates Integer.
@Tag Integer iRef2 = 100;                   // (2) Type context. Annotates Integer.
@Tag java.lang.Integer iRef3 = 100;         // (3) Declaration context.
                                                   Compile-time error!
@Tag int compute() { return (@Tag int) 3.14; }// (4) Type context is int type.
@Tag void compute(int i) {}                   // (5) No return type specified.
                                              //     Compile-time error!
```

```
@Target(ElementType.FIELD)
@interface Tag {}
__________________________________
java.lang. @Tag Integer iRef1 = 100; // (1a) Type context. Compile-time error!
@Tag Integer iRef2 = 100;             // (2a) Declaration context. Annotates iRef2.
@Tag java.lang.Integer iRef3 = 100;   // (3a) Declaration context. Annotates iRef3.
```

```
@Target({ElementType.TYPE_USE, ElementType.FIELD})
@interface Tag {}
----------------------------------
java.lang. @Tag Integer iRef1 = 100; // (1b) Type context. Annotates Integer.
@Tag Integer iRef2 = 100;            // (2b) Declaration and type context.
@Tag java.lang.Integer iRef3 = 100;   // (3b) Declaration context. Annotates iRef3.
```

```
@Target(value = {ElementType.METHOD, ElementType.FIELD,
                 ElementType.METHOD})       // Compile-time error!
@interface Tag {}
```

```
@interface Tag {}                // No target specified.
--------------------------------------------------------------------------
@Tag class Box<@Tag E> {         // Class declaration: OK.
                                 // Type parameter: Compile-time error!
  <@Tag T> void doIt(T t) {}     // Type parameter: Compile-time error!
  void doTask() throws @Tag Exception {}     // Type context: Compile-time error!
  int iTod(double d) { return (@Tag int) d; }// Type context: Compile-time error!
  @Tag boolean flag;                         // Field declaration. OK.
}
```

```
@Target({})                                  // (1) Empty target.
@interface Exclusive {}

@interface ExtraExclusive {
  Exclusive value() default @Exclusive;     // (2) Annotation as element type.
}
______________________________________________________________________________
@Exclusive class Titanium {}                 // (3) Compile-time error!
@ExtraExclusive class Krypton {}             // (4) OK.
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(value = {ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
public @interface MusicMeta {}
```

```
@MusicMeta public class Composition {          // (1) On class
  @MusicMeta private String description;      // (2) On field
  @MusicMeta public void play() {             // (3) On method
//  @MusicMeta int volume;                     // (4) Compile-time error!
  }
}
```

```
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Playable {}
```

```
@MusicMeta
@Playable
public class Composition {}
```

```
public class Song extends Composition {}
```

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented                             // (1)
public @interface Pending {}
```

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Verified {}          // (2)
```

```
@Documented
@Verified
public class Gizmo {}                              // (3)
```

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

```
@Choice(color="Green", size=Size.S)                     // Compile-time error!
@Choice(color="Yellow", size=Size.XL)                   // Compile-time error!
@Choice(color="Red", size=Size.M)                       // Compile-time error!
@Choice(color="White")                                  // Compile-time error!
class Item {}
```

```
@Documented
@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface Repeatable {
  Class<? extends Annotation> value();  // Indicates the containing annot type
}
```

```
import java.lang.annotation.*;
enum Size {S, M, L, XL}
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Choices.class)    // @Repeatable specifies the container
                              // annotation type Choices.
@interface Choice {
  String color();
  Size size() default Size.L;
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  Choice[] value();              // Return value is an array of Choice.
}
```

```
Annotations for 'class Item':
  @Choices(value={@Choice(size=S, color="Green"),
                  @Choice(size=XL, color="Yellow"),
                  @Choice(size=M, color="Red"),
                  @Choice(size=L, color="White")})
...
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  Choice[] value();                  // Return value is an array of Choice.
  double minPrice() default 1.00;    // Must specify a default value.
}
```

```
@Repeatable(Choices.class)        // Compile-time error since invalid return type
                                  // at (1).
@interface Choice {
  String color();
  Size size() default Size.L;
}

@Retention(RetentionPolicy.RUNTIME)
@interface Choices {
  String[] value();                // (1) Invalid return type.
}
```

```
public class A {
  public void doThings() {}
}

public class B extends A {
  @Override
  public void dothings() {}          // Wrong method name.
}
```

```
public boolean equals(Object obj)
```

```
class Gizmo {
  public boolean equals(Gizmo obj) { // (1) Overloaded, as parameter doesn’t match.
    // ...
  }
}
```

```
@Target(ElementType=METHOD)
@Retention(RetentionPolicy=SOURCE)
public @interface Override
```

```
@FunctionalInterface
public interface FIX {       // Compiles without errors.One abstract method.
  void doThings();
}

@FunctionalInterface
public interface FIXIt extends FIX {  // Compiles without errors. Abstract method
}                                     // inherited.

@FunctionalInterface
public interface FIY {       // Compile-time error! More than one abstract method.
  void doThings();
  void doOtherThings();
}

@FunctionalInterface
public interface FIZ {       // Compile-time error! No abstract method.
}
```

```
@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface FunctionalInterface
```

```
String  since()      default "";
boolean forRemoval() default false;
```

```
class LoginHandler {
  /**
   * The field SIGN_IN_ATTEMPTS has now been deprecated,
   * and replaced by more appropriate name.
   * @deprecated Use ACCOUNT_LOCKOUT_THRESHOLD instead.
   */
  @Deprecated(since = "10", forRemoval = false)         // Ordinarily deprecated
  public static final int SIGN_IN_ATTEMPTS = 4;         // static field.
  public static final int ACCOUNT_LOCKOUT_THRESHOLD = 4;

  /**
   * The following method has now been deprecated.
   * @deprecated Use initTwoFactorAuthentication() instead
   */
  @Deprecated(since = "10", forRemoval = true)          // Terminally deprecated
  public void initAuthentication() { /* ...*/ }         // instance method.

  public void initTwoFactorAuthentication() { /* ... */ }
}

public class UserLogin {
  public static void main(String[] args) {
    LoginHandler lh = new LoginHandler();
    lh.initAuthentication();                            // Removal warning
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
  }
}
```

```
>javac -deprecation UserLogin.java
UserLogin.java:24: warning: [removal] initAuthentication() in LoginHandler has
been deprecated and marked for removal
    lh.initAuthentication();                            // Removal warning
      ^
UserLogin.java:25: warning: [deprecation] SIGN_IN_ATTEMPTS in LoginHandler has
been deprecated
    System.out.println(LoginHandler.SIGN_IN_ATTEMPTS);  // Deprecation warning
                                   ^
2 warnings
```

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

```
@Documented
@Retention(RUNTIME)
@Target({CONSTRUCTOR,FIELD,LOCAL_VARIABLE,METHOD,PACKAGE,MODULE,PARAMETER,TYPE})
public @interface Deprecated
```

```
@SuppressWarnings("unchecked")
public class PhoneCenter {

  @SuppressWarnings({"unchecked", "deprecation"})
  public void callLandline() {}

}
```

```
import java.util.ArrayList;
import java.util.List;

@SuppressWarnings(value={"unchecked", "deprecation"})  // (1)
public class ATSuppressWarnings {
  /** Mixing legacy code and generic code. */
  public void undisciplinedMethod() {
    List wordList1 = new ArrayList<String>();  // Assigning parameterized type
                                               // to raw type.
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
    wordList1.add("911");                      // (3) Unchecked call
    wordList2.add("119");                      // OK
  }

  /** Overriding and using a deprecated method. */
  @Override
  public void finalize() throws Throwable { // (4) Overriding a deprecated method.
    super.finalize();                       // (5) Usage of deprecated method.
  }
}
```

```
>javac -Xlint:unchecked -deprecation ATSuppressWarnings.java
ATSuppressWarnings.java:9: warning: [unchecked] unchecked conversion
    List<String> wordList2 = wordList1;        // (2) Unchecked conversion
                             ^
  required: List<String>
  found:    List
ATSuppressWarnings.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    wordList1.add("911");                      // (3) Unchecked call
                 ^
  where E is a type-variable:
    E extends Object declared in interface List
ATSuppressWarnings.java:16: warning: [deprecation] finalize() in Object has been
deprecated
  public void finalize() throws Throwable { // (4) Overriding a deprecated method.
              ^
ATSuppressWarnings.java:17: warning: [deprecation] finalize() in Object has been
deprecated
    super.finalize();                       // (5) Usage of deprecated method.
         ^
4 warnings
```

```
@Target({TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE,MODULE})
@Retention(SOURCE)
public @interface SuppressWarnings
```

```
import java.util.ArrayList;
import java.util.List;
public class SafeVarargsTest{
  @SafeVarargs                                          // (1)
  private static void printList(List<String>... toys) { // (2) List<String>[]
    for (List<String> toy : toys) {
      System.out.println(toy);
    }
  }

  public static void main(String[] args) {
    List<String> tList = new ArrayList<String>();
    tList.add("vaporizer"); tList.add("slime gun");
    printList(tList);                                   // (3) new List<String>[]
  }
}
```

```
>javac SafeVarargsTest.java
Note: SafeVarargsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

>javac -Xlint:unchecked SafeVarargsTest.java
SafeVarargsTest.java:5: warning: [unchecked] Possible heap pollution from parame-
terized vararg type List<String>
  private static void printList(List<String>... toys) { // (2) List<String>[]
                                                ^
SafeVarargsTest.java:14: warning: [unchecked] unchecked generic array creation for
varargs parameter of type List<String>[]
    printList(tList);                                   // (3) new List<String>[]
             ^
2 warnings
```

```
>javac SafeVarargsTest.java
>java SafeVarargsTest
[vaporizer, slime gun]
```

```
Constructor<?>[] cons = classobj.getDeclaredConstructors();
Method[] methods      = classobj.getDeclaredMethods();
Field[] fields        = classobj.getDeclaredFields();
```

```
Annotation[] annotations = annotatedElement.getDeclaredAnnotations();
```

```
TaskInfo[] tias = NuclearPlant.class.getDeclaredAnnotationsByType(TaskInfo.class);
```

```
@Pending
@TaskInfo(
    priority   = TaskInfo.TaskPriority.HIGH,
    taskDesc   = "Class for running a nuclear reactor.",
    assignedTo = {"Tom", "Dick", "Harriet"}
    )
public class NuclearPlant {

  @Pending
  public NuclearPlant() {}

  @Deprecated(forRemoval = true, since = "8")
  public boolean outOfProduction;

  @Deprecated(since = "10")
  public void notInUse() {}

  @Pending
  @TaskInfo(
      taskDesc   = "Procedure for nuclear reactor shutdown",
      assignedTo = {"Tom", "Harriet"}
      )
  public void shutDownNuclearReactor() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Exchange nuclear rods",
      assignedTo = {"Tom", "Dick"}
      )
  public void changeNuclearRods() {}

  @TaskInfo(
      priority   = TaskInfo.TaskPriority.LOW,
      taskDesc   = "Adjust nuclear fuel",
      assignedTo = {"Harriet"}
      )
  public void adjustNuclearFuel() {}

  @TaskInfo(
      taskDesc   = "Start nuclear reactor",
      assignedTo = "Dick"
      )
  public void startNuclearReactor() {}

  @Pending
  @Override
  public String toString() {
    return "TBD";
  }
}
```

```
import static java.lang.System.out;

import java.lang.reflect.AnnotatedElement;
import java.util.Arrays;
import java.util.stream.Stream;

public class TaskInfoAnnotationProcessor {

  public static void printTaskInfoAnnotation(AnnotatedElement... elements) {// (1)
    Stream.of(elements)                                                 // (2)
      .filter(ae -> ae.isAnnotationPresent(TaskInfo.class))             // (3)
      .peek(ae -> out.printf("%s annotation for '%s':%n",               // (4)
          TaskInfo.class.getName(), ae))
      .flatMap(ae -> Stream.of(
                 ae.getDeclaredAnnotationsByType(TaskInfo.class)))    // (5)
      .forEach(a -> {                                                   // (6)
        out.printf("  Task description: %s%n", a.taskDesc());
        out.printf("  Priority: %s%n", a.priority());
        out.printf("  Assigned to: %s%n", Arrays.toString(a.assignedTo()));
      });
  }

  public static void main(String[] args) {
    Class<?> classobj = NuclearPlant.class;                             // (7)
    printTaskInfoAnnotation(classobj);                                  // (8)
    printTaskInfoAnnotation(classobj.getDeclaredMethods());             // (9)
  }
}
```

```
TaskInfo annotation for 'class NuclearPlant':
  Task description: Class for running a nuclear reactor.
  Priority: HIGH
  Assigned to: [Tom, Dick, Harriet]
TaskInfo annotation for 'public void NuclearPlant.shutDownNuclearReactor()':
  Task description: Procedure for nuclear reactor shutdown
  Priority: NORMAL
  Assigned to: [Tom, Harriet]
TaskInfo annotation for 'public void NuclearPlant.changeNuclearRods()':
  Task description: Exchange nuclear rods
  Priority: LOW
  Assigned to: [Tom, Dick]
TaskInfo annotation for 'public void NuclearPlant.adjustNuclearFuel()':
  Task description: Adjust nuclear fuel
  Priority: LOW
  Assigned to: [Harriet]
TaskInfo annotation for 'public void NuclearPlant.startNuclearReactor()':
  Task description: Start nuclear reactor
  Priority: NORMAL
  Assigned to: [Dick]
```

```
import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
import java.util.stream.Stream;

public class AnnotationPrinter {

  public static void printAllAnnotations(Class<?> classobj) {               // (1)
    printAnnotatedElements(classobj);                                       // (2)
    printAnnotatedElements(classobj.getDeclaredConstructors());             // (3)
    printAnnotatedElements(classobj.getDeclaredMethods());                  // (4)
    printAnnotatedElements(classobj.getDeclaredFields());                   // (5)
  }

  public static void printAnnotatedElements(AnnotatedElement... elements) {// (6)
    Stream.of(elements)                                                     // (7)
      .peek(ae -> System.out.printf("Annotations for \'%s\':%n", ae))      // (8)
      .flatMap(ae -> Stream.of(ae.getDeclaredAnnotations()))               // (9)
      .forEach(a -> System.out.println("  " + a));                          // (10)
  }
}
```

```
public class AnnotationClient {
  public static void main(String[] args) {
    AnnotationPrinter.printAllAnnotations(NuclearPlant.class);
  }
}
```

```
Annotations for 'class NuclearPlant':
  @Pending()
  @TaskInfo(priority=HIGH, taskDesc="Class for running a nuclear reactor.",
            assignedTo={"Tom", "Dick", "Harriet"})
Annotations for 'public NuclearPlant()':
  @Pending()
Annotations for 'public void NuclearPlant.shutDownNuclearReactor()':
  @Pending()
  @TaskInfo(priority=NORMAL, taskDesc="Procedure for nuclear reactor shutdown",
            assignedTo={"Tom", "Harriet"})
Annotations for 'public void NuclearPlant.notInUse()':
  @java.lang.Deprecated(forRemoval=false, since="10")
Annotations for 'public void NuclearPlant.changeNuclearRods()':
  @TaskInfo(priority=LOW, taskDesc="Exchange nuclear rods",
            assignedTo={"Tom", "Dick"})
Annotations for 'public void NuclearPlant.adjustNuclearFuel()':
  @TaskInfo(priority=LOW, taskDesc="Adjust nuclear fuel", assignedTo={"Harriet"})
Annotations for 'public void NuclearPlant.startNuclearReactor()':
  @TaskInfo(priority=NORMAL, taskDesc="Start nuclear reactor",
            assignedTo={"Dick"})
Annotations for 'public java.lang.String NuclearPlant.toString()':
  @Pending()
Annotations for 'public boolean NuclearPlant.outOfProduction':
  @java.lang.Deprecated(forRemoval=true, since="8")
```

```
public @interface Test3Annotation { }
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target(PACKAGE)
public @interface Test3Annotation { }
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({FIELD,TYPE})
public @interface Test3Annotation { }
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.*;
@Target({PACKAGE,TYPE_USE})
public @interface Test3Annotation { }
```

```
@interface Test4Annotation {
  int[] value() default {1,2,3};
}
```

```
// (1) INSERT ANNOTATION HERE
public class Test4 { }
```

```
@Test5Annotation
public class Test5<@Test5Annotation(value="a") T> {
  @Test5Annotation({"a","b"})
  private T var;
  public Test5(T var) {
    this.var = var;
  }
}
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE})
@interface Test5Annotation {
  String[] value() default {"x"};
}
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
  String[] value() default "x";
}
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({CONSTRUCTOR, FIELD, TYPE_PARAMETER, TYPE})
@interface Test5Annotation {
  String[] values() default {"x"};
}
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
```

```
@Target({METHOD, FIELD, TYPE_PARAMETER})
@interface Test5Annotation {
  String[] values() default "x";
}
```

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

```
public @interface Location {
  String value() default "London";
  int[] coordinates() default 1;
}
```

```
public @interface Location {
  String value();
  int[] coordinates default {1,1};
}
```

```
public @interface Location {
  String value() default null;
  int[] coordinates default 1;
}
```

```
public @interface Location {
  String value();
  int[] coordinates() default {1,1};
}
```

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Target;
@Target({FIELD, CONSTRUCTOR, TYPE})
public @interface Descriptor {
  String value();
  String[] details() default "";
}
```

```
@Descriptor("Music to play")
public interface Playable { }
public enum Style {
  @Descriptor(value="Rock music", details={"listen"})
  ROCK,
  @Descriptor(value="POP music", details="dance")
  POP,
  @Descriptor(value="Jazz music", details={"listen", "dance"})
  JAZZ,
  @Descriptor("Classic music")
  CLASSIC;
}
```

```
public class Music {
  @Descriptor(value="Music to play")
  Music m = new Music();
}
```

```
@Descriptor("Music to play",details={"listen","dance"})
public class Music {
  Music m = new Music();
}
```

```
public class Player {
  @Descriptor("Music to play")
  public static void main(String[] args) {
    Music m = new Music();
  }
}
```

```
public class Player {
  public static void main(String[] args) {
    @Descriptor(value="Music to play",details={"listen","dance"})
    new Music();
  }
}
```

```
import java.lang.annotation.*;
@Repeatable(Container.class)
public @interface Containee { int value(); }
public @interface Container {
  // (1) INSERT CODE HERE
}
```

```
    String name() default "x";
    Containee value() default @Containee;
```

```
String name() default "x";
Containee[] values() default {@Containee(1), @Containee(2)};
```

```
String name();
Containee[] values() default {@Containee(1), @Containee(2)};
```

```
@Folder("/data")
@Folder(value="/tmp", temp=true)
public class Storage { }
```

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

```
public @interface Folders {
  Folder[] paths();
}
```

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp();
}
```

```
public @interface Folders {
  Folder[] value();
}
```

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String value();
  boolean temp() default false;
}
```

```
  String value();
  boolean temp() default false;
}
```

and

```
public @interface Folders {
  Folder[] value();
}
```

```
import java.lang.annotation.Repeatable;
@Repeatable(Folders.class)
public @interface Folder {
  String path();
  boolean temp() default true;
}
```

```
try {
  String creditCard = "12345678890";
  MessageDigest md = MessageDigest.getInstance("SHA-256");  // (1)
  byte[] digest = md.digest(creditCard.getBytes());         // (2)
  String hash = (new BigInteger(1, digest)).toString(16);   // (3)
} catch (NoSuchAlgorithmException e) {
  e.printStackTrace();
}
```

```
try {
  String creditCard = "12345678890";
  SecretKey key = KeyGenerator.getInstance("AES").generateKey(); // (1)
  Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");       // (2)
  cipher.init(Cipher.ENCRYPT_MODE, key);                         // (3)
  byte[] encryptedValue = cipher.doFinal(creditCard.getBytes()); // (4)
  GCMParameterSpec ps = cipher.getParameters()
                            .getParameterSpec(GCMParameterSpec.class);
  cipher.init(Cipher.DECRYPT_MODE, key, ps);                     // (5)
  byte[] decryptedValue = cipher.doFinal(encryptedValue);        // (6)
} catch (GeneralSecurityException e) {
  e.printStackTrace();
}
```

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement s = connection.createStatement()) {
  String value = "whatever";
  ResultSet rs = s.executeQuery("select * from songs where name like '"
                               + value + "'");                 // (1)
} catch (SQLException e) {
  e.printStackTrace();
}
```

```
public class SyncCounter {
  public int value;                                         // (1)
  public synchronized void increment()  { value++; }
  public synchronized int  getValue()   { return value; }
}
```

```
int x = Integer.MAX_VALUE;
int y = x + 1;                                         // (1)
int z = Math.addExact(x, 1);                           // (2)
```

```
boolean veryLargeNum = Double.isInfinite(1/Double.MIN_VALUE); // (1)
boolean nan = Double.isNaN(0.1/0);                            // (2)
```

```
Optional<String> o = computeOptionalResult(); // (1)
String s = null;
if (o.isPresent()) {                          // (2)
  s = o.get();
}
s = o.orElse("Default");                      // (3)
s = o.orElseThrow();                          // (4)
```

```
public class Authenticator {
  public boolean authenticate() {           // (1)
    /* Perform authentication logic. */
  }
}
```

```
policy.url.1=file:${java.home}/conf/security/java.policy
policy.url.2=file:/anypath/java.policy
```

```
keystore "xyz.keystore";
grant codeBase "file:/application.jar" signedBy "abc" {
  permission java.net.SocketPermission "localhost:8080", "listen";
  permission java.io.FilePermission "/FileSystemPath", "read, write";
};
```

```
SocketPermission socketPermission
        = new SocketPermission("localhost:8080", "listen");     // (1)
FilePermission filePermission
        = new FilePermission("/FileSystemPath", "read, write"); // (2)
try {
  AccessController.checkPermission(socketPermission);           // (3)
  AccessController.checkPermission(filePermission);             // (4)
  // (5) ...
} catch(AccessControlException e) {
  // (6) ...
}
```

```
String content = AccessController.doPrivileged(
  new PrivilegedAction<String>() {
   @Override
  public String run() {                                    // (1)
   String result = null;
   try {
    result = Files.readString(Path.of("/FileSystemPath")); // (2)
   } catch (IOException ex) {
    // ...                                                   (3)
   }
   return result;
  }
 });
```

```
public class RQ3 {
  public static void main(String[] args) {
    int z = Math.addExact(Integer.MAX_VALUE, 1);
    System.out.println(z);
  }
}
```

https://education.oracle.com/java-se-17-developer/pexam_1Z0-829

https://education.oracle.com/java-se-11-developer/pexam_1Z0-819 https://education.oracle.com/oracle-certification-path/product_267 https://coderanch.com/f/24/java-programmer-OCPJP https://enthuware.com/java-certification-mock-exams/oracle-certified-professional/ocp-java-17-exam-1z0-829 https://education.oracle.com/buy-exam

https://education.oracle.com/file/general/ocp_candidate_agreement.pdf

https://education.oracle.com/java-se-17-developer/pexam_1Z0-829

```
public Product(int id, String name, double price) {
...
}
```

```
public Product(int id, String name, double price) {
  this();
}
```

```
public record Product(int id, String name, double price) {
...
}
```

```
return this.x + super.toString() + x;
```

```
    Funky1 p1;                                     //      LHS: String -> void
    p1 = s -> System.out.println(s);               // (1) RHS: String -> void
    p1 = s -> s.length();                          // (2) RHS: String -> int
    p1 = s -> s.toUpperCase();                     // (3) RHS: String -> String
    p1 = s -> { s.toUpperCase(); };                // (4) RHS: String -> void
//  p1 = s -> { return s.toUpperCase(); };         // (5) RHS: String -> String

    Funky2 p2;                                     //      LHS: String -> String
//  p2 = s -> System.out.println(s);               // (6) RHS: String -> void
//  p2 = s -> s.length();                          // (7) RHS: String -> int
    p2 = s -> s.toUpperCase();                     // (8) RHS: String -> String
//  p2 = s -> { s.toUpperCase(); };                // (9) RHS: String -> void
    p2 = s -> { return s.toUpperCase(); };         // (10)RHS: String -> String
```

```
Object obj = () -> System.out.println("Jingle");  // Compile-time error!
```

```
NavigableSet<String> set2 = new TreeSet<>(set1);
```

```
TreeSet<String>(SortedSet<String> set)
```

```
NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
```

```
TreeSet<String>(Collection<? extends String> collection)
```

```
(a,         b)  ->  b  +  a
("X",      "Y") -> "Y" + "X" => "YX"
("YX",     "Z") -> "Z" + "YX" => "ZYX"
("ZYX",    "A") -> "A" + "ZYX" => "AZYX"
("AZYX",   "B") -> "B" + "AZYX" => "BAZYX"
("BAZYX",  "C") -> "C" + "BAZYX" => "CBAZYX"
```

	Pattern	Output
(a)	`.00`	`|.46|`
(b)	`.##`	`|.46|`
(c)	`.0#`	`|.46|`
(d)	`#.00`	`|.46|`
(e)	`#.0#`	`|.46|`
(f)	`#.##`	`|0.46|`
(g)	`.#0`	`Throws java.lang.IllegalArgumentException.`

limits[0]	limits[1]	limits[2]
0	-1	1

formats[0]	formats[1]	formats[2]
"zero"	"negative"	"positive"

```
limits[1] <= 0.9 < limits[2]
   -1        <= 0.9 <     1
```

```
/wrk/./document/../book/../chapter1
/wrk/chapter1
chapter1
./document/../book/../chapter1
./document/../book/..
```

```
Files.copy(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

```
new Thread(new R1(),"|R1a|").run();
new Thread(new R1(),"|R1b|").start();
```

```
class Base {
  protected int i;
  Base() { add(1); }
  void add(int v) { i += v; }
  void print() { System.out.println(i); }
}
class Extension extends Base {
  Extension() { add(2); }
  void add(int v) { i += v*2; }
}
public class Qd073 {
  public static void main(String[] args) {
    bogo(new Extension());
  }
  static void bogo(Base b) {
    b.add(8);
    b.print();
  }
}
```

```
public class Qcb90 {
  private int a;
  private int b;
  public void f() {
    a = 0;
    b = 0;
    int[] c = { 0 };
    g(b, c);
    System.out.println(a + " " + b + " " + c[0] + " ");
  }
  public void g(int b, int[] c) {
    a = 1;
    b = 1;
    c[0] = 1;
  }
  public static void main(String[] args) {
    Qcb90 obj = new Qcb90();
    obj.f();
  }
}
```

```
class A extends I {
  int value;
  void setValue(int val) { value = val; }
  int getValue() { return value; }
}
```

```
abstract class C implements I {
  int getValue() { return 0; }
  abstract void increment();
}
```

```
class E implements I {
  int value;
  public void setValue(int val) { value = val; }
}
```

```
public class Q200A80 {
  public static void main(String[] args) {
    callType(10);
  }

  private static void callType(Number num){
    System.out.println("Number passed");
  }

  private static void callType(Object obj){
    System.out.println("Object passed");
  }
}
```

```
class MyClass {
  public static void main(String[] args) {
    int k = 0;
    int l = 0;
    for (int i = 0; i <= 3; i++) {
      k++;
      if (i == 2) break;
      l++;
    }
    System.out.println(k + ", " + l);
  }
}
```

```
int[][] nums = {{20}, {30}, {40}};
```

```
{
  int sum = 0;
  for (int[] row : nums[])
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

```
{
  int sum = 0;
  for (int[] row : nums[][])
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

```
{
  int sum = 0;
  for (int[] row : nums)
    for (int val : nums[row])
      sum += val;
  System.out.println(sum);
}
```

```
{
  int sum = 0;
  for (int[] row : nums)
    for (int val : row)
      sum += val;
  System.out.println(sum);
}
```

```
{
  int sum = 0;
  for (Integer[] row : nums)
    for (int val : row)
      sum += val;
  System.out.println(sum);
}
```

```
public class Q1408b {
  public static void main(String[] args) {
    int i = 0;
    while (++i == i) {
      System.out.println(i++);
    }
  }
}
```

```
public class RemainderFun {
  public static void main(String[] args) {
    int i = 24, k = 7;
    System.out.print( i %  k + "|");
    System.out.print( i % -k + "|");
    System.out.print(-i %  k + "|");
    System.out.println(-i % -k);
  }
}
```

```
public class Switchy {
  public static void main(String[] args) {
    final String s1 = "January";
    final String yr = " 2022";
    s1.concat(yr);
    switch (s1) {
      default -> System.out.println("Sorry.");
      case "January" + yr, s1 + " 2023" -> System.out.println("OK.");
    }
  }
}
```

```
public class CodeMe {
  public static void main(String[] args) {
    boolean flag = false;
    if (false)          // (1)
      flag = !flag;
    System.out.println(flag);
  }
}
```

```
import java.util.List;

class Person implements Comparable<Person> {
  public int compareTo(Person p) {
    return 1;
  }
}

class Student extends Person {
  public int compareTo(Student s) {
    return 2;
  }
}

public class Calling {
  public static void main(String[] args) {
    Person  p1 = new Person();
    Student s1 = new Student();
    Student s2 = new Student();
    Person  p2 = s1;
    System.out.println(List.of(
        p1.compareTo(s1), p1.compareTo(p2),
        p2.compareTo(s1), p2.compareTo(p1),
        s1.compareTo(p1), s1.compareTo(p2),
        s1.compareTo(s2)));
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class Q12A56 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add(0, "Ada");
    strList.add("Alyla");
    strList.set(strList.size()-1, "Otto");
    strList.add(strList.size()-1, "Anna");
    System.out.println(strList);                       // (1)
    int size = strList.size();
    for (int i = 0; i < size; ++i) {
      strList.add(strList.get(size-1-i));
    }
    System.out.println(strList);                       // (2)
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class Q12A55 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add(strList.size(), "Anna");
    strList.add(strList.size()-1, "Ada");
    strList.add(strList.size()-1, "Otto");
    strList.add(0, "Alyla");
    System.out.println(strList);
    int size = strList.size();
    for (int i = 0; i < size/2; ++i) {
      String strTemp = strList.get(i);
      strList.set(i, strList.get(size-1-i));
      strList.set(size-1-i, strTemp);
    }
    System.out.println(strList);
  }
}
```

```
import java.util.function.Predicate;
public class Test {
  public static void main(String[] args) throws IOException  {
    Stream<String> shapes = Stream.of("Circle","Cube");
    // (1) INSERT CODE HERE
    shapes.filter(p).forEach(v->System.out.println(v));
  }
}
```

```
Predicate<String> p = s -> {
  s.toLowerCase();
  return s.contains("c");
};
```

```
Predicate p = s -> {
  s = s.toString().toLowerCase();
  return s.contains("c");
};
```

```
interface IA            { boolean equals(Object obj); }
interface IB extends IA { boolean doIt(String str); }
interface IC extends IB { boolean doIt(String str); }
interface ID extends IC { boolean equals(Object obj);}
```

```
import java.time.LocalDate;

public class Q11A35 {
  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2015, 1, 1);
    date.withYear(5);
    System.out.println(date.plusMonths(12));
  }
}
```

```
import java.time.*;
import java.time.format.*;

public class Q11A95 {
  public static void main(String[] args) {
    String inputStr = "The time is 15 minutes past 10PM.";
    String pattern = "'The time is 'm' minutes past 'ha.";
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern(pattern);
    // (1) INSERT CODE HERE
  }
}
```

```
Runnable r = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
Thread t = new Thread(r);
t.start();
```

```
public class Q100A82 {
  public static void main(String[] args) {
    Object o = choose(991, "800");                                 // (1)
    Number n1 = choose(991, 3.14);                                 // (2)
    Number n2 = Q100A82.<Double>choose((double)991, 3.14);         // (3)
    int k = (int) choose(1.3, 3.14);                               // (4)
    int l = (int) (double) choose(1.3, 3.14);                      // (5)
  }

  public static <T extends Comparable<T>> T choose(T t1, T t2) {
    return t1.compareTo(t2) >= 0 ? t1 : t2;
  }
}
```

```
Thread t = new Thread() {
  public void start() {
    Work.doIt();
  }
};
t.start();
```

```
Runnable r = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
r.start();
```

```
Thread t = new Thread(new Work());
t.start();
```

```
Runnable t = new Runnable() {
  public void run() {
    Work.doIt();
  }
};
t.run();
```

```
public class Outer {
  private int a = 1;
  private int b = 1;
  private int c = 1;

  class Inner {
    private int a = 2;

    int get() {
      int c = 3;
      // (1) INSERT CODE HERE
      return c;
    }
  }

  Outer() {
    Inner i = new Inner();
    System.out.println(i.get());
  }

  public static void main(String[] args) {
    new Outer();
  }
}
```

```
public enum FrequentFlyer {
  PLATINUM(20), GOLD(10), SILVER(5), BASIC(0);
  private double extra;

  FrequentFlyer(double extra) {
    this.extra = extra;
  }

  public static FrequentFlyer max(FrequentFlyer c1, FrequentFlyer c2) {
    return c1.compareTo(c2) < 0 ? c2 : c1;
  }

  public static FrequentFlyer max2(FrequentFlyer c1, FrequentFlyer c2) {
    return c1.extra < c2.extra ? c2 : c1;
  }

  public static void main (String[] args) {
    System.out.println(GOLD.ordinal() > SILVER.ordinal());
    System.out.println(max(GOLD, SILVER));
    System.out.println(max2(GOLD, SILVER));
  }
}
```

```
public class Vertical {
  private int alt;
  public synchronized void up() {
    ++alt;
  }
  public void down() {
    --alt;
  }
  public synchronized void jump() {
    int a = alt;
    up();
    down();
    System.out.println(a == alt);
  }
}
```

```
interface Wagger{}
class Pet implements Wagger{}
class Dog extends Pet {}
class Cat extends Pet {}
```

```
public class Q100A51 {
  public static void main(String[] args) {
    List<Pet> p = new ArrayList<>();
    List<Dog> d = new ArrayList<>();
    List<Cat> c = new ArrayList<>();
    examine(p);
    examine(d);
    examine(c);
  }

  static void examine(/* INSERT PARAMETER TYPE HERE */ pets) {      // (1)
    System.out.print("Your pets need urgent attention.");
  }
}
```

```
package p1;
public enum Format {
  JPEG, GIF, TIFF;
}
```

```
package p1;
public class Util {
  public enum Format {
    JPEG { public String toString() {return "Jpeggy"; }},
    GIF  { public String toString() {return "Giffy"; }},
    TIFF { public String toString() {return "Tiffy"; }};
  }
  public static <T> void print(T t) {
    System.out.print("|" + t + "|");
  }
}
```

```
import static p1.Util.Format;
import static p1.Util.print;

public class Importing {
  static final int JPEG = 200;
  public static void main(String[] args) {
    final int JPEG = 100;
    print(JPEG);
    // (1) INSERT CODE HERE
  }
}
```

```
import java.util.ArrayList;
import java.util.List;
public class Q400A70 {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(2019); list.add(2020); list.add(2021);
    System.out.println("Before: " + list);
    for (int i : list) {
      int index = list.indexOf(i);
      list.set(index, ++i);
    }
    System.out.println("After:  " + list);
  }
}
```

```
public class Person {
  protected transient String name;
  Person() { this.name = "NoName"; }
  Person(String name) { this.name = name; }
}
```

```
public class Student extends Person {
  protected long studNum;
  Student() { }
  Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
}
```

```
import java.io.Serializable;
public class GraduateStudent extends Student implements Serializable {
  private int year;
  GraduateStudent(String name, long studNum, int year) {
    super(name, studNum);
    this.year = year;
  }

  public String toString() {
    return "(" + name + ", " + studNum + ", " + year + ")";
  }
}
```

```
import java.io.*;
public class Q800A60 {
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      GraduateStudent stud1 = new GraduateStudent("Aesop", 100, 1);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      GraduateStudent stud2 = (GraduateStudent) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

```
public class SupA<T> {
  public List<?> fuddle() { return null; }
  public List scuddle(T t) { return null; }
}
public class SubB<U> extends SupA<U> {
  public List fuddle() { return null;}
  public List<?> scuddle(U t) { return null; }
}
public class SubC<V> extends SupA<V> {
  public List<V> fuddle() { return null;}
  public List<? extends Object> scuddle(V t) { return null; }
}
```

```
package p2;
enum March {LEFT, RIGHT;                                  // (1)
  public String toString() {
    return "Top-level enum";
  }
}
public class DefenceInDepth {
  public enum March {LEFT, RIGHT;                         // (2)
    public String toString() {
      return "Static enum";
    }
  }
  public enum Military { INFANTRY, AIRFORCE;
    public static enum March {LEFT, RIGHT;                // (3)
```

```
      public String toString() {
        return "Statically nested enum";
      }
    }
  }
}
```

```
package p3;
// (4) INSERT IMPORTS HERE
public class MarchingOrders {
  public static void main(String[] args) {
    System.out.println(March.LEFT);
    System.out.println(DefenceInDepth.March.LEFT);
    System.out.println(p2.DefenceInDepth.March.LEFT);
    System.out.println(Military.March.LEFT);
    System.out.println(DefenceInDepth.Military.March.LEFT);
    System.out.println(p2.DefenceInDepth.Military.March.LEFT);
    System.out.println(LEFT);
  }
}
```

```
public class A {
  public A() {}
  public A(int i) { this(); }
}
public class B extends A {
  public boolean B(String msg) { return false; }
}
public class C extends B {
  private C() { super(); }
```

```
import p2.*;
import p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
import p2.DefenceInDepth;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
import static p2.DefenceInDepth;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March.LEFT;
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.*;
import p2.*;
import static p2.DefenceInDepth.*;
import static p2.DefenceInDepth.Military.March;
```

```
  public C(String msg) { this(); }
  public C(int i) {}
}
```

```
public enum Scale5 {
  GOOD, BETTER, BEST;

  public char getGrade() {
    return switch (this) {
      case GOOD   -> 'C';
      case BETTER -> 'B';
      case BEST   -> 'A';
    };
  }

  public static void main (String[] args) {
    System.out.println(GOOD.getGrade());
  }
}
```

```
// File: MyClass.java
package p4;
class MySuperclass {
  public        Integer step1(int i)                     { return 1; }     // (1)
  protected     String  step2(String str1, String str2) { return str1; }  // (2)
  public        String  step2(String str1)               { return str1; }  // (3)
  public static String  step2()                          { return "Hi"; }  // (4)

  public MyClass      makeIt()  { return new MyClass(); }                  // (5)
  public MySuperclass makeIt2() { return new MyClass(); }                  // (6)
}

public class MyClass extends MySuperclass {
  // (7) INSERT METHOD DECLARATION HERE
}
```

```
public class Q800A60 {
  static void printFirst(Integer... ints) {
    System.out.println("Integer...: " + ints[0]);
  }

  static void printFirst(Number... nums) {
    System.out.println("Number...: " + nums[0]);
  }

  static void printFirst(Object... objs) {
    System.out.println("Object...: " + objs[0]);
  }

  public static void main(String[] args) {
    printFirst(10);
    printFirst((byte)20);
    printFirst('3', '0');
    printFirst("40");
    printFirst((short)50, 55);
    printFirst((Number[])new Integer[] {70, 75});
  }
}
```

```
public class Initialization {
  private static String msg(String msg) {
    System.out.println(msg);
    return msg;
  }
  static String m = msg("1");
  { m = msg("2"); }
  static { m = msg("3"); }
  public static void main(String[] args) {
    Object obj = new Initialization();
  }
}
```

```
# File: MyResources_en_US.properties
greeting = Long time no see!
```

```
import java.util.*;
public class LocatingBundlesForDefaultLocale {
  public static void main(String[] args) {
    Locale norLocale = new Locale("no", "NO");
    ResourceBundle rbs = ResourceBundle.getBundle("MyResources", norLocale);
    System.out.println(rbs.getString("greeting"));
    Locale.setDefault(norLocale);
    System.out.println(rbs.getString("greeting"));
  }
}
```

```
public class Syncher2 {
  static final int[] intArray = new int[2];

  private static void pause() {
    while (intArray[0] == 0) {
      try { intArray.wait(); }
      catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    }
  }

  public static void main (String[] args) {
    Thread runner = new Thread() {
      public void run() {
        synchronized (intArray) {
          pause();
          System.out.println(intArray[0] + intArray[1]);
    }}};

    runner.start();
    intArray[0] = intArray[1] = 10;
    synchronized(intArray) {
      intArray.notify();
    }
  }
}
```

```
public class Thingy<T> implements Comparable<T> {
  private T value;
  public Thingy(T value) {
    this.value = value;
  }
  public String toString() {
    return value.toString();
  }
  public int compareTo(T obj) {
    return this.value.toString().compareTo(obj.toString());
  }
}
```

```
import java.math.*;
import java.util.*;
public class TestThingy {
  public static void main(String[] args) {
    Thingy[] values  = {
      new Thingy<BigDecimal>(BigDecimal.valueOf(12.99)),
      new Thingy<BigDecimal>(BigDecimal.valueOf(7.99)),
      new Thingy<BigDecimal>(BigDecimal.valueOf(7)),
      new Thingy<BigDecimal>(BigDecimal.valueOf(9.99))};
    Arrays.sort(values);
    for (Thingy t: values) {
      System.out.print(t + " ");
    }
  }
}
```

```
public class Widget<T> {
  private T value;
  private static int result;
  public Widget(T value) { this.value = value; }
  public void compute(Comparable<T> c) {
    result += c.compareTo(value);
  }
  public static int getResult() { return result; }
}
```

```
import java.time.*;
public class TestWidget {
  public static void main(String[] args) {
    Widget<String> t1 = new Widget<>("ACME");
    t1.compute(v -> v.length());
    t1.compute(v -> v.indexOf("C"));
    Widget<LocalDate> t2 = new Widget<>(LocalDate.of(2020, 10, 20));
    t2.compute(v -> v.getDayOfMonth());
    System.out.println(Widget.getResult());
  }
}
```

```
import java.util.*;
public class Test13A10 {
  public static void main(String[] args) {
    Integer[] values = {4, 2, 6, 3, 5};
    Arrays.sort(values, (x, y) -> x - y);
    System.out.println(Arrays.toString(values));
    }
  }
}
```

```
import java.time.*;
import java.time.format.DateTimeFormatter;
public class Q700A12 {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(0, 1, 1);
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd MM yy G");
    System.out.println(d.format(dtf));
  }
}
```

```
Comparable<?>[] values = { LocalDate.of(2022, Month.JANUARY, 1),
    LocalDate.of(2022, Month.FEBRUARY, 5),
    Integer.valueOf(3),
    LocalDateTime.of(2022, Month.JANUARY, 22, 1, 12, 2),
    "4"};
long result = Arrays.stream(values)
    .mapToInt(v -> (v instanceof LocalDateTime ldt)
                    ? ldt.toLocalTime().getSecond() : 1)
    .sum();
System.out.println(result);
```

```
String txt = """
  a
    b
  c
  """;
txt.lines().map(String::length).forEachOrdered(System.out::print);
```

```
import java.nio.file.*;
public class Test {
  public static void main(String[] args) {
    Path p1 = Path.of("/users/joe");
    Path p2 = Path.of("/users/bob");
    Path p3 = p1.resolve(p1.relativize(p2));
    Path p4 = p3.normalize();
    System.out.println(p3.getName(1) + " " + p4.getName(1));
  }
}
```

```
import java.util.*;
public class Localizing {  public static void main(String[] args) {
    ResourceBundle rb = ResourceBundle.getBundle("resources", new Locale("en"));
    DateTimeFormatter fmt = DateTimeFormatter.ofPattern(rb.getString("f1"),
                                                         new Locale("en","GB"));
    LocalDate d = LocalDate.of(2022, 4, 1);
    System.out.println(fmt.format(d));
  }
}
```

```
// File: resources.properties
f1=yy-MMM-dd
```

```
// File: resources_en.properties
f2=MMM-dd-yy
```

```
// File: resources_en_GB.properties
f1=dd-MMM-yy
```

```
public class TestX {
  public static void main(String[] args) {
    System.out.println(action(Integer.valueOf("1")));
  }
  public static boolean action(Object obj) {
    return (obj instanceof String value1 && value1.contains("1") ||
            obj instanceof Integer value2 && value2.intValue() < 1);
    }
}
```

```
import java.time.Duration;
public record Song(String title, Duration duration) {
  // (1) INSERT CODE HERE
}
```

```
public class TestY {
  public static void main(String[] args) {
    Song song = new Song("Imagine", 106);
    System.out.println(song);
  }
}
```

```
public Song(int seconds) {
  this(this.title.toUpperCase(), Duration.ofSeconds(seconds));
}
```

```
public Song(int seconds) {
  this(title.toUpperCase(), Duration.ofSeconds(seconds));
}
```

```
public Song(String title, int seconds) {
  this(title.toUpperCase(), Duration.ofSeconds(seconds));
}
```

```
public Song(String title, int seconds) {
  this.title = title.toUpperCase();
  this.duration = Duration.ofSeconds(seconds);
}
```

```
public Song {
  this.title.toUpperCase();
  this.duration = Duration.ofSeconds(seconds));
}
```

```
./Sun/1_Mercury
./Sun/2_Venus
./Sun/3_Earth
./Sun/3_Earth/1_Moon
./Sun/4_Mars
./Sun/4_Mars/1_Phobos
./Sun/4_Mars/2_Deimos
./Sun/5_Jupiter
./Sun/5_Jupiter/1_Io
./Sun/5_Jupiter/2_Europa
./Sun/5_Jupiter/3_Ganymede
./Sun/5_Jupiter/4_Calisto
./Sun/6_Saturn
./Sun/7_Uranus
./Sun/8_Neptune
```

```
try {
  Path sun = Path.of("./Sun");
  Files.walk(sun)
       .map(p->p.getName(p.getNameCount()-1).toString())
       .sorted()
       .limit(3)
       .forEach(p->System.out.println(p.substring(2)));
} catch (IOException e) {
  e.printStackTrace();
}
```

```
try {
  Path test = Path.of("data.txt");
  Map<String, List<String>> values
      = Files.lines(test).collect(Collectors.groupingBy(
                                       ln -> ln.substring(ln.indexOf(":") + 1)));
  System.out.println(values);
} catch (IOException e) {
  e.printStackTrace();
}
```

```
module travel {
  requires transport;
  uses transport.mode.TransportType;
}
```

```
module basic {
  requires travel;
  provides travel.mode.TransportType with basic.mode.Horse;
}
```

```
module basic {
  requires transport.mode;
  provides TransportType with Horse;
}
```

```
module basic {
  requires transport;
  provides transport.mode.TransportType with basic.mode.Horse;
}
```

```
module basic {
  requires travel.mode;
  provides TransportType with Horse;
}
```

```
[Ada]
[Ada, Alyla]
[Ada, Otto]
[Ada, Anna, Otto]
[Ada, Anna, Otto, Otto, Anna, Ada]
```

```
                     0      1     2
                   [2019, 2020, 2021]
After iteration 1: [2020, 2020, 2021]
After iteration 2: [2021, 2020, 2021]
After iteration 3: [2022, 2020, 2021]
```

```
printFirst(10);           // new Integer[] {Integer.valueOf(10)}
printFirst((byte)20);     // new Number[] {Byte.valueOf(20)}
printFirst('3', '0');     // new Object[] {Character.valueOf('3'),
                          //               Character.valueOf('0')}
printFirst("40");         // new Object[] {"40"}
printFirst((short)50, 55);// new Number[] {Short.valueOf(50),
                          //               Integer.valueOf(55)}
printFirst((Number[])new Integer[] {70, 75}); // Passed as array of Number
```

```
p1: /users/joe
p2: /users/bob
p1.relativize(p2):                   ../bob
p3 = p1.resolve(p1.relativize(p2)): /users/joe/../bob
p4 = p3.normalize():                 /users/bob
```

```
java -Djava.util.logging.config.file=another-config
```

```
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit   = 50000
java.util.logging.FileHandler.count   = 1
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.formatter    = java.util.logging.XMLFormatter
java.util.logging.ConsoleHandler.level = WARNING
```

```
package org.foo;
import java.util.logging.*;
public class Test {
  private static Logger logger = Logger.getLogger(org.foo.Test.class.getName());
  public static void main(String[] args) {
  ...
  }
}
```

```
package org.foo;
import java.util.logging.*;
```

```
package org.foo;
import java.util.logging.*;
public class Test {
  private static Logger logger = Logger.getLogger(org.foo.Test.class.getName());
  public static void main(String[] args) {
    try {
      // any actions that may potentially throw an exception
    }catch(Exception e) {
      logger.log(Level.ERROR, "Some error message", e);
    }
    logger.log(Level.INFO, "Some other message");
  }
}
```

```
logger.log(Level.FINE, "This is " + someVar + " value");
```

```
if(logger.isLoggable(Level.FINER)) {                              // (1)
  logger.log(Level.FINE, "This is " + someVar + " value");
}
```

```
logger.log(Level.FINE, "This is {0} value", someVar);         // (2)
```